Guide: Racket

Version 5.1

Matthew Flatt,
Robert Bruce Findler,
and PLT

February 14, 2011

This guide is intended for programmers who are new to Racket or new to some part of Racket. It assumes programming experience, so if you are new to programming, consider instead reading *How to Design Programs*. If you want an especially quick introduction to Racket, start with ***Quick***: *An Introduction to Racket with Pictures*.

Chapter 2 provides a brief introduction to Racket. From Chapter 3 on, this guide dives into details—covering much of the Racket toolbox, but leaving precise details to ***Reference***: *Racket* and other reference manuals.

Contents

5

1   Welcome to Racket

Depending on how you look at it, **Racket** is

- a *programming language*—a dialect of Lisp and a descendant of Scheme;
- a *family* of programming languages—variants of Racket, and more; or
- a set of *tools*—for using a family of programming languages.

Where there is no room for confusion, we use simply *Racket*.

Racket's main tools are

- `racket`, the core compiler, interpreter, and run-time system;
- **DrRacket**, the programming environment; and
- `raco`, a command-line tool for executing **Ra**cket **co**mmands that install packages, build libraries, and more.

Most likely, you'll want to explore the Racket language using DrRacket, especially at the beginning. If you prefer, you can also work with the command-line `racket` interpreter and your favorite text editor. The rest of this guide presents the language mostly independent of your choice of editor.

If you're using DrRacket, you'll need to choose the proper language, because DrRacket accommodates many different variants of Racket, as well as other languages. Assuming that you've never used DrRacket before, start it up, type the line

```
#lang racket
```

in DrRacket's top text area, and then click the Run button that's above the text area. DrRacket then understands that you mean to work in the normal variant of Racket (as opposed to the smaller `racket/base` or many other possibilities).

If you've used DrRacket before with something other than a program that starts #lang, DrRacket will remember the last language that you used, instead of inferring the language from the #lang line. In that case, use the Language|Choose Language... menu item. In the dialog that appears, select the first item, which tells DrRacket to use the language that is declared in a source program via #lang. Put the #lang line above in the top text area, still.

1.1 Interacting with Racket

DrRacket's bottom text area and the `racket` command-line program (when started with no options) both act as a kind of calculator. You type a Racket expression, hit the Return key, and the answer is printed. In the terminology of Racket, this kind of calculator is called a *read-eval-print loop* or *REPL*.

A number by itself is an expression, and the answer is just the number:

```
> 5
5
```

A string is also an expression that evaluates to itself. A string is written with double quotes at the start and end of the string:

```
> "Hello, world!"
"Hello, world!"
```

Racket uses parentheses to wrap larger expressions—almost any kind of expression, other than simple constants. For example, a function call is written: open parenthesis, function name, argument expression, and closing parenthesis. The following expression calls the built-in function `substring` with the arguments `"the boy out of the country"`, `4`, and `7`:

```
> (substring "the boy out of the country" 4 7)
"boy"
```

1.2 Definitions and Interactions

You can define your own functions that work like `substring` by using the `define` form, like this:

```
(define (extract str)
  (substring str 4 7))

> (extract "the boy out of the country")
"boy"
> (extract "the country out of the boy")
"cou"
```

Although you can evaluate the `define` form in the REPL, definitions are normally a part of a program that you want to keep and use later. So, in DrRacket, you'd normally put the definition in the top text area—called the *definitions area*—along with the `#lang` prefix:

```
#lang racket
```

```
(define (extract str)
  (substring str 4 7))
```

If calling (extract "the boy") is part of the main action of your program, that would go in the definitions area, too. But if it was just an example expression that you were using to explore extract, then you'd more likely leave the definitions area as above, click Run, and then evaluate (extract "the boy") in the REPL.

When using command-line racket instead of DrRacket, you'd save the above text in a file using your favorite editor. If you save it as "extract.rkt", then after starting racket in the same directory, you'd evaluate the following sequence:

```
> (enter! "extract.rkt")
> (extract "the gal out of the city")
"gal"
```

The enter! form both loads the code and switches the evaluation context to the inside of the module, just like DrRacket's Run button.

1.3   Creating Executables

If your file (or definitions area in DrRacket) contains

```
#lang racket

(define (extract str)
  (substring str 4 7))

(extract "the cat out of the bag")
```

then it is a complete program that prints "cat" when run. To package this program as an executable, choose one of the following options:

- In DrRacket, you can select the Racket|Create Executable... menu item.

- From a command-line prompt, run raco exe ⟨*src-filename*⟩, where ⟨*src-filename*⟩ contains the program. See §2 "raco exe: Creating Stand-Alone Executables" for more information.

- With Unix or Mac OS X, you can turn the program file into an executable script by inserting the line

  ```
  #! /usr/bin/env racket
  ```

  at the very beginning of the file. Also, change the file permissions to executable using chmod +x ⟨*filename*⟩ on the command line.

See §19.2 "Scripts" for more information on script files.

The script works as long as `racket` is in the user's executable search path. Alternately, use a full path to `racket` after `#!` (with a space between `#!` and the path), in which case the user's executable search path does not matter.

1.4   A Note to Readers with Lisp/Scheme Experience

If you already know something about Racket or Lisp, you might be tempted to put just

```
(define (extract str)
  (substring str 4 7))
```

into `"extract.rktl"` and run `racket` with

```
> (load "extract.rktl")
> (extract "the dog out")
"dog"
```

That will work, because `racket` is willing to imitate a traditional Lisp environment, but we strongly recommend against using `load` or writing programs outside of a module.

Writing definitions outside of a module leads to bad error messages, bad performance, and awkward scripting to combine and run programs. The problems are not specific to `racket`; they're fundamental limitations of the traditional top-level environment, which Scheme and Lisp implementations have historically fought with ad hoc command-line flags, compiler directives, and build tools. The module system is designed to avoid these problems, so start with `#lang`, and you'll be happier with Racket in the long run.

2   Racket Essentials

This chapter provides a quick introduction to Racket as background for the rest of the guide. Readers with some Racket experience can safely skip to §3 "Built-In Datatypes".

2.1   Simple Values

Racket values include numbers, booleans, strings, and byte strings. In DrRacket and documentation examples (when you read the documentation in color), value expressions are shown in green.

Numbers are written in the usual way, including fractions and imaginary numbers:

```
1       3.14
1/2     6.02e+23
1+2i    99999999999999999999999
```

Booleans are `#t` for true and `#f` for false. In conditionals, however, all non-`#f` values are treated as true.

Strings are written between doublequotes. Within a string, backslash is an escaping character; for example, a backslash followed by a doublequote includes a literal doublequote in the string. Except for an unescaped doublequote or backslash, any Unicode character can appear in a string constant.

```
"Hello, world!"
"Benjamin \"Bugsy\" Siegel"
"λx:(μα.α→α).xx"
```

When a constant is evaluated in the REPL, it typically prints the same as its input syntax. In some cases, the printed form is a normalized version of the input syntax. In documentation and in DrRacket's REPL, results are printed in blue instead of green to highlight the difference between an input expression and a printed result.

Examples:
```
> 1.0000
1.0
> "Bugs \u0022Figaro\u0022 Bunny"
"Bugs \"Figaro\" Bunny"
```

2.2   Simple Definitions and Expressions

A program module is written as

`#lang` ⟨*langname*⟩ ⟨*topform*⟩*

where a ⟨*topform*⟩ is either a ⟨*definition*⟩ or an ⟨*expr*⟩. The REPL also evaluates ⟨*topform*⟩s.

In syntax specifications, text with a gray background, such as `#lang`, represents literal text. Whitespace must appear between such literals and nonterminals like ⟨*id*⟩, except that whitespace is not required before or after `(`, `)`, `[`, or `]`. A comment, which starts with `;` and runs until the end of the line, is treated the same as whitespace.

Following the usual conventions, * in a grammar means zero or more repetitions of the preceding element, + means one or more repetitions of the preceding element, and { } groups a sequence as an element for repetition.

2.2.1 Definitions

A definition of the form

  `(` `define` ⟨*id*⟩ ⟨*expr*⟩ `)`

binds ⟨*id*⟩ to the result of ⟨*expr*⟩, while

  `(` `define` `(` ⟨*id*⟩ ⟨*id*⟩`*` `)` ⟨*expr*⟩⁺ `)`

binds the first ⟨*id*⟩ to a function (also called a *procedure*) that takes arguments as named by the remaining ⟨*id*⟩s. In the function case, the ⟨*expr*⟩s are the body of the function. When the function is called, it returns the result of the last ⟨*expr*⟩.

Examples:
```
(define pie 3)             ; defines pie to be 3
(define (piece str)        ; defines piece as a function
  (substring str 0 pie))   ;  of one argument
> pie
3
> (piece "key lime")
"key"
```

Under the hood, a function definition is really the same as a non-function definition, and a function name does not have to be used in a function call. A function is just another kind of value, though the printed form is necessarily less complete than the printed form of a number or string.

Examples:
```
> piece
#<procedure:piece>
> substring
#<procedure:substring>
```

19

A function definition can include multiple expressions for the function's body. In that case, only the value of the last expression is returned when the function is called. The other expressions are evaluated only for some side-effect, such as printing.

Examples:
```
(define (bake flavor)
  (printf "pre-heating oven...\n")
  (string-append flavor " pie"))
> (bake "apple")
pre-heating oven...
"apple pie"
```

Racket programmers prefer to avoid side-effects, so a definition usually has just one expression in its body. It's important, though, to understand that multiple expressions are allowed in a definition body, because it explains why the following `nobake` function fails to include its argument in its result:

```
(define (nobake flavor)
  string-append flavor "jello")

> (nobake "green")
"jello"
```

Within `nobake`, there are no parentheses around `string-append flavor "jello"`, so they are three separate expressions instead of one function-call expression. The expressions `string-append` and `flavor` are evaluated, but the results are never used. Instead, the result of the function is just the result of the final expression, `"jello"`.

2.2.2 An Aside on Indenting Code

Line breaks and indentation are not significant for parsing Racket programs, but most Racket programmers use a standard set of conventions to make code more readable. For example, the body of a definition is typically indented under the first line of the definition. Identifiers are written immediately after an open parenthesis with no extra space, and closing parentheses never go on their own line.

DrRacket automatically indents according to the standard style when you type Enter in a program or REPL expression. For example, if you hit Enter after typing `(define (greet name)`, then DrRacket automatically inserts two spaces for the next line. If you change a region of code, you can select it in DrRacket and hit Tab, and DrRacket will re-indent the code (without inserting any line breaks). Editors like Emacs offer a Racket or Scheme mode with similar indentation support.

Re-indenting not only makes the code easier to read, it gives you extra feedback that your parentheses match in the way that you intended. For example, if you leave out a closing

parenthesis after the last argument to a function, automatic indentation starts the next line under the first argument, instead of under the `define` keyword:

```
(define (halfbake flavor
                  (string-append flavor " creme brulee")))
```

In this case, indentation helps highlight the mistake. In other cases, where the indentation may be normal while an open parenthesis has no matching close parenthesis, both `racket` and DrRacket use the source's indentation to suggest where a parenthesis might be missing.

2.2.3 Identifiers

Racket's syntax for identifiers is especially liberal. Excluding the special characters

```
( ) [ ] { } " , ' ` ; # | \
```

and except for the sequences of characters that make number constants, almost any sequence of non-whitespace characters forms an ⟨*id*⟩. For example `substring` is an identifier. Also, `string-append` and `a+b` are identifiers, as opposed to arithmetic expressions. Here are several more examples:

```
+
Hfuhruhurr
integer?
pass/fail
john-jacob-jingleheimer-schmidt
a-b-c+1-2-3
```

2.2.4 Function Calls (Procedure Applications)

We have already seen many function calls, which are called *procedure applications* in more traditional terminology. The syntax of a function call is

```
( ⟨id⟩ ⟨expr⟩* )
```

where the number of ⟨*expr*⟩s determines the number of arguments supplied to the function named by ⟨*id*⟩.

The `racket` language pre-defines many function identifiers, such as `substring` and `string-append`. More examples are below.

In example Racket code throughout the documentation, uses of pre-defined names are hyperlinked to the reference manual. So, you can click on an identifier to get full details about its use.

```
> (string-append "rope" "twine" "yarn")  ; append strings
"ropetwineyarn"
> (substring "corduroys" 0 4)            ; extract a substring
"cord"
> (string-length "shoelace")             ; get a string's length
8
> (string? "Ceci n'est pas une string.") ; recognize strings
#t
> (string? 1)
#f
> (sqrt 16)                              ; find a square root
4
> (sqrt -16)
0+4i
> (+ 1 2)                                ; add numbers
3
> (- 2 1)                                ; subtract numbers
1
> (< 2 1)                                ; compare numbers
#f
> (>= 2 1)
#t
> (number? "c'est une number")           ; recognize numbers
#f
> (number? 1)
#t
> (equal? 6 "half dozen")                ; compare anything
#f
> (equal? 6 6)
#t
> (equal? "half dozen" "half dozen")
#t
```

2.2.5 Conditionals with `if`, `and`, `or`, and `cond`

The next simplest kind of expression is an `if` conditional:

```
( if ⟨expr⟩ ⟨expr⟩ ⟨expr⟩ )
```

The first ⟨expr⟩ is always evaluated. If it produces a non-#f value, then the second ⟨expr⟩ is evaluated for the result of the whole `if` expression, otherwise the third ⟨expr⟩ is evaluated for the result.

Example:
```
> (if (> 2 3)
```

```
      "bigger"
      "smaller")
"smaller"

(define (reply s)
  (if (equal? "hello" (substring s 0 5))
      "hi!"
      "huh?"))

> (reply "hello racket")
"hi!"
> (reply "λx:(μα.α→α).xx")
"huh?"
```

Complex conditionals can be formed by nesting `if` expressions. For example, you could
make the `reply` function work when given non-strings:

```
(define (reply s)
  (if (string? s)
      (if (equal? "hello" (substring s 0 5))
          "hi!"
          "huh?")
      "huh?"))
```

Instead of duplicating the `"huh?"` case, this function is better written as

```
(define (reply s)
  (if (if (string? s)
          (equal? "hello" (substring s 0 5))
          #f)
      "hi!"
      "huh?"))
```

but these kinds of nested `if`s are difficult to read. Racket provides more readable shortcuts
through the `and` and `or` forms, which work with any number of expressions:

```
( and ⟨expr⟩* )
( or  ⟨expr⟩* )
```

The `and` form short-circuits: it stops and returns `#f` when an expression produces `#f`, other-
wise it keeps going. The `or` form similarly short-circuits when it encounters a true result.

Examples:
```
(define (reply s)
  (if (and (string? s)
           (>= (string-length s) 5)
           (equal? "hello" (substring s 0 5)))
```

```
      "hi!"
      "huh?"))
> (reply "hello racket")
"hi!"
> (reply 17)
"huh?"
```

Another common pattern of nested `if`s involves a sequence of tests, each with its own result:

```
(define (reply-more s)
  (if (equal? "hello" (substring s 0 5))
      "hi!"
      (if (equal? "goodbye" (substring s 0 7))
          "bye!"
          (if (equal? "?" (substring s (- (string-length s) 1)))
              "I don't know"
              "huh?"))))
```

The shorthand for a sequence of tests is the `cond` form:

```
( cond {[ ⟨expr⟩ ⟨expr⟩* ]}* )
```

A `cond` form contains a sequence of clauses between square brackets. In each clause, the first ⟨expr⟩ is a test expression. If it produces true, then the clause's remaining ⟨expr⟩s are evaluated, and the last one in the clause provides the answer for the entire `cond` expression; the rest of the clauses are ignored. If the test ⟨expr⟩ produces `#f`, then the clause's remaining ⟨expr⟩s are ignored, and evaluation continues with the next clause. The last clause can use `else` as a synonym for a `#t` test expression.

Using `cond`, the `reply-more` function can be more clearly written as follows:

```
(define (reply-more s)
  (cond
   [(equal? "hello" (substring s 0 5))
    "hi!"]
   [(equal? "goodbye" (substring s 0 7))
    "bye!"]
   [(equal? "?" (substring s (- (string-length s) 1)))
    "I don't know"]
   [else "huh?"]))

> (reply-more "hello racket")
"hi!"
> (reply-more "goodbye cruel world")
"bye!"
> (reply-more "what is your favorite color?")
"I don't know"
```

24

```
> (reply-more "mine is lime green")
"huh?"
```

The use of square brackets for `cond` clauses is a convention. In Racket, parentheses and square brackets are actually interchangable, as long as `(` is matched with `)` and `[` is matched with `]`. Using square brackets in a few key places makes Racket code even more readable.

2.2.6 Function Calls, Again

In our earlier grammar of function calls, we oversimplified. The actual syntax of a function call allows an arbitrary expression for the function, instead of just an ⟨*id*⟩:

   `(` ⟨*expr*⟩ ⟨*expr*⟩* `)`

The first ⟨*expr*⟩ is often an ⟨*id*⟩, such as `string-append` or `+`, but it can be anything that evaluates to a function. For example, it can be a conditional expression:

```
(define (double v)
  ((if (string? v) string-append +) v v))

> (double "mnah")
"mnahmnah"
> (double 5)
10
```

Syntactically, the first expression in a function call could even be a number—but that leads to an error, since a number is not a function.

```
> (1 2 3 4)
procedure application: expected procedure, given: 1;
arguments were: 2 3 4
```

When you accidentally omit a function name or when you use parentheses around an expression, you'll most often get an "expected a procedure" error like this one.

2.2.7 Anonymous Functions with `lambda`

Programming in Racket would be tedious if you had to name all of your numbers. Instead of writing `(+ 1 2)`, you'd have to write

```
> (define a 1)
> (define b 2)
> (+ a b)
3
```

25

It turns out that having to name all your functions can be tedious, too. For example, you might have a function `twice` that takes a function and an argument. Using `twice` is convenient if you already have a name for the function, such as `sqrt`:

```
(define (twice f v)
  (f (f v)))

> (twice sqrt 16)
2
```

If you want to call a function that is not yet defined, you could define it, and then pass it to `twice`:

```
(define (louder s)
  (string-append s "!"))

> (twice louder "hello")
"hello!!"
```

But if the call to `twice` is the only place where `louder` is used, it's a shame to have to write a whole definition. In Racket, you can use a `lambda` expression to produce a function directly. The `lambda` form is followed by identifiers for the function's arguments, and then the function's body expressions:

```
( lambda ( ⟨id⟩* ) ⟨expr⟩+ )
```

Evaluating a `lambda` form by itself produces a function:

```
> (lambda (s) (string-append s "!"))
#<procedure>
```

Using `lambda`, the above call to `twice` can be re-written as

```
> (twice (lambda (s) (string-append s "!"))
         "hello")
"hello!!"
> (twice (lambda (s) (string-append s "?!"))
         "hello")
"hello?!?!"
```

Another use of `lambda` is as a result for a function that generates functions:

```
(define (make-add-suffix s2)
  (lambda (s) (string-append s s2)))

> (twice (make-add-suffix "!") "hello")
"hello!!"
```

```
> (twice (make-add-suffix "?!") "hello")
"hello?!?!"
> (twice (make-add-suffix "...") "hello")
"hello......"
```

Racket is a *lexically scoped* language, which means that `s2` in the function returned by `make-add-suffix` always refers to the argument for the call that created the function. In other words, the `lambda`-generated function "remembers" the right `s2`:

```
> (define louder (make-add-suffix "!"))
> (define less-sure (make-add-suffix "?"))
> (twice less-sure "really")
"really??"
> (twice louder "really")
"really!!"
```

We have so far referred to definitions of the form (`define` ⟨*id*⟩ ⟨*expr*⟩) as "non-function definitions." This characterization is misleading, because the ⟨*expr*⟩ could be a `lambda` form, in which case the definition is equivalent to using the "function" definition form. For example, the following two definitions of `louder` are equivalent:

```
(define (louder s)
  (string-append s "!"))

(define louder
  (lambda (s)
    (string-append s "!")))

> louder
#<procedure:louder>
```

Note that the expression for `louder` in the second case is an "anonymous" function written with `lambda`, but, if possible, the compiler infers a name, anyway, to make printing and error reporting as informative as possible.

2.2.8 Local Binding with `define`, `let`, and `let*`

It's time to retract another simplification in our grammar of Racket. In the body of a function, definitions can appear before the body expressions:

(`define` (⟨*id*⟩ ⟨*id*⟩*) ⟨*definition*⟩* ⟨*expr*⟩+)
(`lambda` (⟨*id*⟩*) ⟨*definition*⟩* ⟨*expr*⟩+)

Definitions at the start of a function body are local to the function body.

Examples:

```
(define (converse s)
  (define (starts? s2) ; local to converse
    (define len2 (string-length s2))  ; local to starts?
    (and (>= (string-length s) len2)
         (equal? s2 (substring s 0 len2))))
  (cond
    [(starts? "hello") "hi!"]
    [(starts? "goodbye") "bye!"]
    [else "huh?"]))
> (converse "hello!")
"hi!"
> (converse "urp")
"huh?"
> starts? ; outside of converse, so...
reference to undefined identifier: starts?
```

Another way to create local bindings is the `let` form. An advantage of `let` is that it can be used in any expression position. Also, `let` binds many identifiers at once, instead of requiring a separate `define` for each identifier.

$$(\ \texttt{let} \ (\ \{[\ \langle id \rangle \ \langle expr \rangle \ ]\}^* \ ) \ \langle expr \rangle^+ \ )$$

Each binding clause is an ⟨*id*⟩ and a ⟨*expr*⟩ surrounded by square brackets, and the expressions after the clauses are the body of the `let`. In each clause, the ⟨*id*⟩ is bound to the result of the ⟨*expr*⟩ for use in the body.

```
> (let ([x (random 4)]
        [o (random 4)])
    (cond
      [(> x o) "X wins"]
      [(> o x) "O wins"]
      [else "cat's game"]))
"X wins"
```

The bindings of a `let` form are available only in the body of the `let`, so the binding clauses cannot refer to each other. The `let*` form, in contrast, allows later clauses to use earlier bindings:

```
> (let* ([x (random 4)]
         [o (random 4)]
         [diff (number->string (abs (- x o)))])
    (cond
      [(> x o) (string-append "X wins by " diff)]
      [(> o x) (string-append "O wins by " diff)]
      [else "cat's game"]))
"O wins by 1"
```

28

2.3   Lists, Iteration, and Recursion

Racket is a dialect of the language Lisp, whose name originally stood for "LISt Processor."
The built-in list datatype remains a prominent feature of the language.

The `list` function takes any number of values and returns a list containing the values:

```
> (list "red" "green" "blue")
'("red" "green" "blue")
> (list 1 2 3 4 5)
'(1 2 3 4 5)
```

As you can see, a list result prints in the REPL as a quote ' and then a pair of parentheses
wrapped around the printed form of the list elements. There's an opportunity for confusion
here, because parentheses are used for both expressions, such as `(list "red" "green"
"blue")`, and printed results, such as `'("red" "green" "blue")`. In addition to the
quote, parentheses for results are printed in blue in the documentation and in DrRacket,
whereas parentheses for expressions are brown.

A list usually prints with ', but the printed form of a list depends on its content. See §3.8 "Pairs and Lists" for more information.

Many predefined functions operate on lists. Here are a few examples:

```
> (length (list "hop" "skip" "jump"))       ; count the elements
3
> (list-ref (list "hop" "skip" "jump") 0)   ; extract by position
"hop"
> (list-ref (list "hop" "skip" "jump") 1)
"skip"
> (append (list "hop" "skip") (list "jump")) ; combine lists
'("hop" "skip" "jump")
> (reverse (list "hop" "skip" "jump"))       ; reverse order
'("jump" "skip" "hop")
> (member "fall" (list "hop" "skip" "jump")) ; check for an element
#f
```

2.3.1   Predefined List Loops

In addition to simple operations like `append`, Racket includes functions that iterate over the
elements of a list. These iteration functions play a role similar to `for` in Java, Racket, and
other languages. The body of a Racket iteration is packaged into a function to be applied to
each element, so the `lambda` form becomes particularly handy in combination with iteration
functions.

Different list-iteration functions combine iteration results in different ways. The `map` func-
tion uses the per-element results to create a new list:

```
> (map sqrt (list 1 4 9 16))
'(1 2 3 4)
> (map (lambda (i)
         (string-append i "!"))
       (list "peanuts" "popcorn" "crackerjack"))
'("peanuts!" "popcorn!" "crackerjack!")
```

The `andmap` and `ormap` functions combine the results by `anding` or `oring`:

```
> (andmap string? (list "a" "b" "c"))
#t
> (andmap string? (list "a" "b" 6))
#f
> (ormap number? (list "a" "b" 6))
#t
```

The `filter` function keeps elements for which the body result is true, and discards elements for which it is `#f`:

```
> (filter string? (list "a" "b" 6))
'("a" "b")
> (filter positive? (list 1 -2 6 7 0))
'(1 6 7)
```

The `map`, `andmap`, `ormap`, and `filter` functions can all handle multiple lists, instead of just a single list. The lists must all have the same length, and the given function must accept one argument for each list:

```
> (map (lambda (s n) (substring s 0 n))
       (list "peanuts" "popcorn" "crackerjack")
       (list 6 3 7))
'("peanut" "pop" "cracker")
```

The `foldl` function generalizes some iteration functions. It uses the per-element function to both process an element and combine it with the "current" value, so the per-element function takes an extra first argument. Also, a starting "current" value must be provided before the lists:

```
> (foldl (lambda (elem v)
           (+ v (* elem elem)))
         0
         '(1 2 3))
14
```

Despite its generality, `foldl` is not as popular as the other functions. One reason is that `map`, `ormap`, `andmap`, and `filter` cover the most common kinds of list loops.

Racket provides a general *list comprehension* form `for/list`, which builds a list by iterating through *sequences*. List comprehensions and related iteration forms are described in see §11 "Iterations and Comprehensions".

2.3.2 List Iteration from Scratch

Although `map` and other iteration functions predefined, they are not primitive in any interesting sense. You can write equivalent iterations using a handful of list primitives.

Since a Racket list is a linked list, the two core operations on a non-empty list are

- `first`: get the first thing in the list; and

- `rest`: get the rest of the list.

Examples:
```
> (first (list 1 2 3))
1
> (rest (list 1 2 3))
'(2 3)
```

To create a new node for a linked list—that is, to add to the front of the list—use the `cons` function, which is short for "construct." To get an empty list to start with, use the `empty` constant:

```
> empty
'()
> (cons "head" empty)
'("head")
> (cons "dead" (cons "head" empty))
'("dead" "head")
```

To process a list, you need to be able to distinguish empty lists from non-empty lists, because `first` and `rest` work only on non-empty lists. The `empty?` function detects empty lists, and `cons?` detects non-empty lists:

```
> (empty? empty)
#t
> (empty? (cons "head" empty))
#f
> (cons? empty)
#f
> (cons? (cons "head" empty))
#t
```

With these pieces, you can write your own versions of the `length` function, `map` function, and more.

Examples:
```
(define (my-length lst)
  (cond
   [(empty? lst) 0]
   [else (+ 1 (my-length (rest lst)))]))
> (my-length empty)
0
> (my-length (list "a" "b" "c"))
3

(define (my-map f lst)
  (cond
   [(empty? lst) empty]
   [else (cons (f (first lst))
               (my-map f (rest lst)))]))

> (my-map string-upcase (list "ready" "set" "go"))
'("READY" "SET" "GO")
```

If the derivation of the above definitions is mysterious to you, consider reading *How to Design Programs*. If you are merely suspicious of the use of recursive calls instead of a looping construct, then read on.

2.3.3   Tail Recursion

Both the `my-length` and `my-map` functions run in $O(n)$ time for a list of length n. This is easy to see by imagining how `(my-length (list "a" "b" "c"))` must evaluate:

```
(my-length (list "a" "b" "c"))
= (+ 1 (my-length (list "b" "c")))
= (+ 1 (+ 1 (my-length (list "c"))))
= (+ 1 (+ 1 (+ 1 (my-length (list)))))
= (+ 1 (+ 1 (+ 1 0)))
= (+ 1 (+ 1 1))
= (+ 1 2)
= 3
```

For a list with n elements, evaluation will stack up n `(+ 1 ...)` additions, and then finally add them up when the list is exhausted.

You can avoid piling up additions by adding along the way. To accumulate a length this way, we need a function that takes both a list and the length of the list seen so far; the code below

uses a local function `iter` that accumulates the length in an argument `len`:

```
(define (my-length lst)
  ; local function iter:
  (define (iter lst len)
    (cond
      [(empty? lst) len]
      [else (iter (rest lst) (+ len 1))]))
  ; body of my-length calls iter:
  (iter lst 0))
```

Now evaluation looks like this:

```
(my-length (list "a" "b" "c"))
= (iter (list "a" "b" "c") 0)
= (iter (list "b" "c") 1)
= (iter (list "c") 2)
= (iter (list) 3)
3
```

The revised `my-length` runs in constant space, just as the evaluation steps above suggest. That is, when the result of a function call, like `(iter (list "b" "c") 1)`, is exactly the result of some other function call, like `(iter (list "c") 2)`, then the first one doesn't have to wait around for the second one, because that takes up space for no good reason.

This evaluation behavior is sometimes called *tail-call optimization*, but it's not merely an "optimization" in Racket; it's a guarantee about the way the code will run. More precisely, an expression in *tail position* with respect to another expression does not take extra computation space over the other expression.

In the case of `my-map`, $O(n)$ space complexity is reasonable, since it has to generate a result of size $O(n)$. Nevertheless, you can reduce the constant factor by accumulating the result list. The only catch is that the accumulated list will be backwards, so you'll have to reverse it at the very end:

Attempting to reduce a constant factor like this is usually not worthwhile, as discussed below.

```
(define (my-map f lst)
  (define (iter lst backward-result)
    (cond
      [(empty? lst) (reverse backward-result)]
      [else (iter (rest lst)
                  (cons (f (first lst))
                        backward-result))]))
  (iter lst empty))
```

It turns out that if you write

```
(define (my-map f lst)
```

```
(for/list ([i lst])
  (f i)))
```

then the `for/list` form in the function both is expanded to essentially the same code as the
`iter` local definition and use. The difference is merely syntactic convenience.

2.3.4   Recursion versus Iteration

The `my-length` and `my-map` examples demonstrate that iteration is just a special case of
recursion. In many languages, it's important to try to fit as many computations as possible
into iteration form. Otherwise, performance will be bad, and moderately large inputs can
lead to stack overflow. Similarly, in Racket, it is sometimes important to make sure that tail
recursion is used to avoid $O(n)$ space consumption when the computation is easily performed
in constant space.

At the same time, recursion does not lead to particularly bad performance in Racket, and
there is no such thing as stack overflow; you can run out of memory if a computation involves
too much context, but exhausting memory typically requires orders of magnitude deeper
recursion than would trigger a stack overflow in other languages. These considerations,
combined with the fact that tail-recursive programs automatically run the same as a loop,
lead Racket programmers to embrace recursive forms rather than avoid them.

Suppose, for example, that you want to remove consecutive duplicates from a list. While
such a function can be written as a loop that remembers the previous element for each itera-
tion, a Racket programmer would more likely just write the following:

```
(define (remove-dups l)
  (cond
    [(empty? l) empty]
    [(empty? (rest l)) l]
    [else
     (let ([i (first l)])
       (if (equal? i (first (rest l)))
           (remove-dups (rest l))
           (cons i (remove-dups (rest l)))))]))

> (remove-dups (list "a" "b" "b" "b" "c" "c"))
'("a" "b" "c")
```

In general, this function consumes $O(n)$ space for an input list of length n, but that's fine,
since it produces an $O(n)$ result. If the input list happens to be mostly consecutive duplicates,
then the resulting list can be much smaller than $O(n)$—and `remove-dups` will also use much
less than $O(n)$ space! The reason is that when the function discards duplicates, it returns the
result of a `remove-dups` call directly, so the tail-call "optimization" kicks in:

```
(remove-dups (list "a" "b" "b" "b" "b" "b"))
= (cons "a" (remove-dups (list "b" "b" "b" "b" "b")))
= (cons "a" (remove-dups (list "b" "b" "b" "b")))
= (cons "a" (remove-dups (list "b" "b" "b")))
= (cons "a" (remove-dups (list "b" "b")))
= (cons "a" (remove-dups (list "b")))
= (cons "a" (list "b"))
= (list "a" "b")
```

2.4   Pairs, Lists, and Racket Syntax

The `cons` function actually accepts any two values, not just a list for the second argument. When the second argument is not `empty` and not itself produced by `cons`, the result prints in a special way. The two values joined with `cons` are printed between parentheses, but with a dot (i.e., a period surrounded by whitespace) in between:

```
> (cons 1 2)
'(1 . 2)
> (cons "banana" "split")
'("banana" . "split")
```

Thus, a value produced by `cons` is not always a list. In general, the result of `cons` is a *pair*. The more traditional name for the `cons?` function is `pair?`, and we'll use the traditional name from now on.

The name `rest` also makes less sense for non-list pairs; the more traditional names for `first` and `rest` are `car` and `cdr`, respectively. (Granted, the traditional names are also nonsense. Just remember that "a" comes before "d," and `cdr` is pronounced "could-er.")

Examples:
```
> (car (cons 1 2))
1
> (cdr (cons 1 2))
2
> (pair? empty)
#f
> (pair? (cons 1 2))
#t
> (pair? (list 1 2 3))
#t
```

Racket's pair datatype and its relation to lists is essentially a historical curiosity, along with the dot notation for printing and the funny names `car` and `cdr`. Pairs are deeply wired into to the culture, specification, and implementation of Racket, however, so they survive in the language.

You are perhaps most likely to encounter a non-list pair when making a mistake, such as accidentally reversing the arguments to `cons`:

```
> (cons (list 2 3) 1)
'((2 3) . 1)
> (cons 1 (list 2 3))
'(1 2 3)
```

Non-list pairs are used intentionally, sometimes. For example, the `make-hash` function takes a list of pairs, where the `car` of each pair is a key and the `cdr` is an arbitrary value.

The only thing more confusing to new Racketeers than non-list pairs is the printing convention for pairs where the second element *is* a pair, but *is not* a list:

```
> (cons 0 (cons 1 2))
'(0 1 . 2)
```

In general, the rule for printing a pair is as follows: use the dot notation always, but if the dot is immediately followed by an open parenthesis, then remove the dot, the open parenthesis, and the matching close parenthesis. Thus, `'(0 . (1 . 2))` becomes `'(0 1 . 2)`, and `'(1 . (2 . (3 . ())))` becomes `'(1 2 3)`.

2.4.1 Quoting Pairs and Symbols with `quote`

A list prints with a quote mark before it, but if an element of a list is itself a list, then no quote mark is printed for the inner list:

```
> (list (list 1) (list 2 3) (list 4))
'((1) (2 3) (4))
```

For nested lists, especially, the `quote` form lets you write a list as an expression in essentially the same way that the list it prints:

```
> (quote ("red" "green" "blue"))
'("red" "green" "blue")
> (quote ((1) (2 3) (4)))
'((1) (2 4) (4))
> (quote ())
'()
```

The `quote` form works with the dot notation, too, whether the quoted form is normalized by the dot-parenthesis elimination rule or not:

```
> (quote (1 . 2))
'(1 . 2)
> (quote (0 . (1 . 2)))
```

36

```
'(0 1 . 2)
```

Naturally, lists of any kind can be nested:

```
> (list (list 1 2 3) 5 (list "a" "b" "c"))
'((1 2 3) 5 ("a" "b" "c"))
> (quote ((1 2 3) 5 ("a" "b" "c")))
'((1 2 3) 5 ("a" "b" "c"))
```

If you wrap an identifier with `quote`, then you get output that looks like an identifier, but with a `'` prefix:

```
> (quote jane-doe)
'jane-doe
```

A value that prints like a quoted identifier is a *symbol*. In the same way that parenthesized output should not be confused with expressions, a printed symbol should not be confused with an identifier. In particular, the symbol `(quote map)` has nothing to do with the `map` identifier or the predefined function that is bound to `map`, except that the symbol and the identifier happen to be made up of the same letters.

Indeed, the intrinsic value of a symbol is nothing more than its character content. In this sense, symbols and strings are almost the same thing, and the main difference is how they print. The functions `symbol->string` and `string->symbol` convert between them.

Examples:
```
> map
#<procedure:map>
> (quote map)
'map
> (symbol? (quote map))
#t
> (symbol? map)
#f
> (procedure? map)
#t
> (string->symbol "map")
'map
> (symbol->string (quote map))
"map"
```

In the same way that `quote` for a list automatically applies itself to nested lists, `quote` on a parenthesized sequence of identifiers automatically applies itself to the identifiers to create a list of symbols:

```
> (car (quote (road map)))
'road
```

```
> (symbol? (car (quote (road map))))
#t
```

When a symbol is inside a list that is printed with ', the ' on the symbol is omitted, since '
is doing the job already:

```
> (quote (road map))
'(road map)
```

The quote form has no effect on a literal expression such as a number or string:

```
> (quote 42)
42
> (quote "on the record")
"on the record"
```

2.4.2 Abbreviating quote with '

As you may have guessed, you can abbreviate a use of quote by just putting ' in front of a
form to quote:

```
> '(1 2 3)
'(1 2 3)
> 'road
'road
> '((1 2 3) road ("a" "b" "c"))
'((1 2 3) road ("a" "b" "c"))
```

In the documentation, ' within an expression is printed in green along with the form after
it, since the combination is an expression that is a constant. In DrRacket, only the ' is
colored green. DrRacket is more precisely correct, because the meaning of quote can vary
depending on the context of an expression. In the documentation, however, we routinely
assume that standard bindings are in scope, and so we paint quoted forms in green for extra
clarity.

A ' expands to a quote form in quite a literal way. You can see this if you put a ' in front
of a form that has a ':

```
> (car ''road)
'quote
> (car '(quote road))
'quote
```

The ' abbreviation works in output as well as input. The REPL's printer recognizes the
symbol 'quote as the first element of a two-element list when printing output, in which
case it uses ' to print the output:

```
> (quote (quote road))
''road
> '(quote road)
''road
> ''road
''road
```

2.4.3 Lists and Racket Syntax

Now that you know the truth about pairs and lists, and now that you've seen `quote`, you're
ready to understand the main way in which we have been simplifying Racket's true syntax.

The syntax of Racket is not defined directly in terms of character streams. Instead, the syntax
is determined by two layers:

- a *reader* layer, which turns a sequence of characters into lists, symbols, and other
  constants; and

- an *expander* layer, which processes the lists, symbols, and other constants to parse
  them as an expression.

The rules for printing and reading go together. For example, a list is printed with parentheses,
and reading a pair of parentheses produces a list. Similarly, a non-list pair is printed with the
dot notation, and a dot on input effectively runs the dot-notation rules in reverse to obtain a
pair.

One consequence of the read layer for expressions is that you can use the dot notation in
expressions that are not quoted forms:

```
> (+ 1 . (2))
3
```

This works because `(+ 1 . (2))` is just another way of writing `(+ 1 2)`. It is practically
never a good idea to write application expressions using this dot notation; it's just a conse-
quence of the way Racket's syntax is defined.

Normally, `.` is allowed by the reader only with a parenthesized sequence, and only before the
last element of the sequence. However, a pair of `.`s can also appear around a single element
in a parenthesized sequence, as long as the element is not first or last. Such a pair triggers a
reader conversion that moves the element between `.`s to the front of the list. The conversion
enables a kind of general infix notation:

```
> (1 . < . 2)
#t
> '(1 . < . 2)
```

```
'(< 1 2)
```

This two-dot convention is non-traditional, and it has essentially nothing to do with the dot notation for non-list pairs. Racket programmers use the infix convention sparingly—mostly for asymmetric binary operators such as `<` and `is-a?`.

3 Built-In Datatypes

The previous chapter introduced some of Racket's built-in datatypes: numbers, booleans, strings, lists, and procedures. This section provides a more complete coverage of the built-in datatypes for simple forms of data.

3.1 Booleans

Racket has two distinguished constants to represent boolean values: `#t` for true and `#f` for false. Uppercase `#T` and `#F` are parsed as the same values, but the lowercase forms are preferred.

The `boolean?` procedure recognizes the two boolean constants. In the result of a test expression for `if`, `cond`, `and`, `or`, etc., however, any value other than `#f` counts as true.

Examples:
```
> (= 2 (+ 1 1))
#t
> (boolean? #t)
#t
> (boolean? #f)
#t
> (boolean? "no")
#f
> (if "no" 1 0)
1
```

3.2 Numbers

A Racket *number* is either exact or inexact:

- An *exact* number is either

  - an arbitrarily large or small integer, such as `5`, `9999999999999999`, or `-17`;

  - a rational that is exactly the ratio of two arbitrarily small or large integers, such as `1/2`, `9999999999999999/2`, or `-3/4`; or

  - a complex number with exact real and imaginary parts (where the imaginary part is not zero), such as `1+2i` or `1/2+3/4i`.

- An *inexact* number is either

- an IEEE floating-point representation of a number, such as `2.0` or `3.14e+87`, where the IEEE infinities and not-a-number are written `+inf.0`, `-inf.0`, and `+nan.0` (or `-nan.0`); or

- a complex number with real and imaginary parts that are IEEE floating-point representations, such as `2.0+3.0i` or `-inf.0+nan.0i`; as a special case, an inexact complex number can have an exact zero real part with an inexact imaginary part.

Inexact numbers print with a decimal point or exponent specifier, and exact numbers print as integers and fractions. The same conventions apply for reading number constants, but `#e` or `#i` can prefix a number to force its parsing as an exact or inexact number. The prefixes `#b`, `#o`, and `#x` specify binary, octal, and hexadecimal interpretation of digits.

§12.6.3 "Reading Numbers" in *Reference: Racket* documents the fine points of the syntax of numbers.

Examples:
```
> 0.5
0.5
> #e0.5
1/2
> #x03BB
955
```

Computations that involve an inexact number produce inexact results, so that inexactness acts as a kind of taint on numbers. Beware, however, that Racket offers no "inexact booleans," so computations that branch on the comparison of inexact numbers can nevertheless produce exact results. The procedures `exact->inexact` and `inexact->exact` convert between the two types of numbers.

Examples:
```
> (/ 1 2)
1/2
> (/ 1 2.0)
0.5
> (if (= 3.0 2.999) 1 2)
2
> (inexact->exact 0.1)
3602879701896397/36028797018963968
```

Inexact results are also produced by procedures such as `sqrt`, `log`, and `sin` when an exact result would require representing real numbers that are not rational. Racket can represent only rational numbers and complex numbers with rational parts.

Examples:
```
> (sin 0)   ; rational...
0
> (sin 1/2) ; not rational...
0.479425538604203
```

In terms of performance, computations with small integers are typically the fastest, where "small" means that the number fits into one bit less than the machine's word-sized representation for signed numbers. Computation with very large exact integers or with non-integer exact numbers can be much more expensive than computation with inexact numbers.

```
(define (sigma f a b)
  (if (= a b)
      0
      (+ (f a) (sigma f (+ a 1) b))))

> (time (round (sigma (lambda (x) (/ 1 x)) 1 2000)))
cpu time: 179 real time: 203 gc time: 16
8
> (time (round (sigma (lambda (x) (/ 1.0 x)) 1 2000)))
cpu time: 0 real time: 1 gc time: 0
8.0
```

The number categories *integer*, *rational*, *real* (always rational), and *complex* are defined in the usual way, and are recognized by the procedures `integer?`, `rational?`, `real?`, and `complex?`, in addition to the generic `number?`. A few mathematical procedures accept only real numbers, but most implement standard extensions to complex numbers.

Examples:
```
> (integer? 5)
#t
> (complex? 5)
#t
> (integer? 5.0)
#t
> (integer? 1+2i)
#f
> (complex? 1+2i)
#t
> (complex? 1.0+2.0i)
#t
> (abs -5)
5
> (abs -5+2i)
abs: expects argument of type <real number>; given -5+2i
> (sin -5+2i)
3.6076607742131563+1.0288031496599335i
```

The `=` procedure compares numbers for numerical equality. If it is given both inexact and exact numbers to compare, it essentially converts the inexact numbers to exact before comparing. The `eqv?` (and therefore `equal?`) procedure, in contrast, compares numbers considering both exactness and numerical equality.

Examples:

```
> (= 1 1.0)
#t
> (eqv? 1 1.0)
#f
```

Beware of comparisons involving inexact numbers, which by their nature can have surprising behavior. Even apparently simple inexact numbers may not mean what you think they mean; for example, while a base-2 IEEE floating-point number can represent 1/2 exactly, it can only approximate 1/10:

Examples:

```
> (= 1/2 0.5)
#t
> (= 1/10 0.1)
#f
> (inexact->exact 0.1)
3602879701896397/36028797018963968
```

3.3 Characters

A Racket *character* corresponds to a Unicode *scalar value*. Roughly, a scalar value is an unsigned integer whose representation fits into 21 bits, and that maps to some notion of a natural-language character or piece of a character. Technically, a scalar value is a simpler notion than the concept called a "character" in the Unicode standard, but it's an approximation that works well for many purposes. For example, any accented Roman letter can be represented as a scalar value, as can any common Chinese character.

Although each Racket character corresponds to an integer, the character datatype is separate from numbers. The `char->integer` and `integer->char` procedures convert between scalar-value numbers and the corresponding character.

A printable character normally prints as `#\` followed by the represented character. An unprintable character normally prints as `#\u` followed by the scalar value as hexadecimal number. A few characters are printed specially; for example, the space and linefeed characters print as `#\space` and `#\newline`, respectively.

Examples:

```
> (integer->char 65)
#\A
> (char->integer #\A)
65
> #\λ
#\λ
> #\u03BB
```

44

```
#\λ
> (integer->char 17)
#\u0011
> (char->integer #\space)
32
```

The `display` procedure directly writes a character to the current output port (see §8 "Input and Output"), in contrast to the character-constant syntax used to print a character result.

Examples:
```
> #\A
#\A
> (display #\A)
A
```

Racket provides several classification and conversion procedures on characters. Beware, however, that conversions on some Unicode characters work as a human would expect only when they are in a string (e.g., upcasing "ß" or downcasing "Σ").

Examples:
```
> (char-alphabetic? #\A)
#t
> (char-numeric? #\0)
#t
> (char-whitespace? #\newline)
#t
> (char-downcase #\A)
#\a
> (char-upcase #\ß)
#\ß
```

The `char=?` procedure compares two or more characters, and `char-ci=?` compares characters ignoring case. The `eqv?` and `equal?` procedures behave the same as `char=?` on characters; use `char=?` when you want to more specifically declare that the values being compared are characters.

Examples:
```
> (char=? #\a #\A)
#f
> (char-ci=? #\a #\A)
#t
> (eqv? #\a #\A)
#f
```

§3.5 "Characters" in ***Reference: Racket*** provides more on characters and character procedures.

45

3.4   Strings (Unicode)

A *string* is a fixed-length array of characters. It prints using doublequotes, where doublequote and backslash characters within the string are escaped with backslashes. Other common string escapes are supported, including `\n` for a linefeed, `\r` for a carriage return, octal escapes using `\` followed by up to three octal digits, and hexadecimal escapes with `\u` (up to four digits). Unprintable characters in a string are normally shown with `\u` when the string is printed.

§12.6.6 "Reading
Strings" in
Reference: *Racket*
documents the fine
points of the syntax
of strings.

The `display` procedure directly writes the characters of a string to the current output port (see §8 "Input and Output"), in contrast to the string-constant syntax used to print a string result.

Examples:
```
> "Apple"
"Apple"
> "\u03BB"
"λ"
> (display "Apple")
Apple
> (display "a \"quoted\" thing")
a "quoted" thing
> (display "two\nlines")
two
lines
> (display "\u03BB")
λ
```

A string can be mutable or immutable; strings written directly as expressions are immutable, but most other strings are mutable. The `make-string` procedure creates a mutable string given a length and optional fill character. The `string-ref` procedure accesses a character from a string (with 0-based indexing); the `string-set!` procedure changes a character in a mutable string.

Examples:
```
> (string-ref "Apple" 0)
#\A
> (define s (make-string 5 #\.))
> s
"....."
> (string-set! s 2 #\λ)
> s
"..λ.."
```

String ordering and case operations are generally *locale-independent*; that is, they work the same for all users. A few *locale-dependent* operations are provided that allow the way

that strings are case-folded and sorted to depend on the end-user's locale. If you're sorting strings, for example, use `string<?` or `string-ci<?` if the sort result should be consistent across machines and users, but use `string-locale<?` or `string-locale-ci<?` if the sort is purely to order strings for an end user.

Examples:
```
> (string<? "apple" "Banana")
#f
> (string-ci<? "apple" "Banana")
#t
> (string-upcase "Straße")
"STRASSE"
> (parameterize ([current-locale "C"])
    (string-locale-upcase "Straße"))
"STRAßE"
```

For working with plain ASCII, working with raw bytes, or encoding/decoding Unicode strings as bytes, use byte strings.

3.5 Bytes and Byte Strings

A *byte* is an exact integer between 0 and 255, inclusive. The `byte?` predicate recognizes numbers that represent bytes.

Examples:
```
> (byte? 0)
#t
> (byte? 256)
#f
```

A *byte string* is similar to a string—see §3.4 "Strings (Unicode)"—but its content is a sequence of bytes instead of characters. Byte strings can be used in applications that process pure ASCII instead of Unicode text. The printed form of a byte string supports such uses in particular, because a byte string prints like the ASCII decoding of the byte string, but prefixed with a `#`. Unprintable ASCII characters or non-ASCII bytes in the byte string are written with octal notation.

Examples:
```
> #"Apple"
#"Apple"
> (bytes-ref #"Apple" 0)
65
> (make-bytes 3 65)
#"AAA"
> (define b (make-bytes 2 0))
```

```
> b
#"\0\0"
> (bytes-set! b 0 1)
> (bytes-set! b 1 255)
> b
#"\1\377"
```

The `display` form of a byte string writes its raw bytes to the current output port (see §8 "Input and Output"). Technically, `display` of a normal (i.e,. character) string prints the UTF-8 encoding of the string to the current output port, since output is ultimately defined in terms of bytes; `display` of a byte string, however, writes the raw bytes with no encoding. Along the same lines, when this documentation shows output, it technically shows the UTF-8-decoded form of the output.

Examples:
```
> (display #"Apple")
Apple
> (display "\316\273")  ; same as "Î»"
Î»
> (display #"\316\273") ; UTF-8 encoding of λ
λ
```

For explicitly converting between strings and byte strings, Racket supports three kinds of encodings directly: UTF-8, Latin-1, and the current locale's encoding. General facilities for byte-to-byte conversions (especially to and from UTF-8) fill the gap to support arbitrary string encodings.

Examples:
```
> (bytes->string/utf-8 #"\316\273")
"λ"
> (bytes->string/latin-1 #"\316\273")
"Î»"
> (parameterize ([current-locale "C"])  ; C locale supports ASCII,
    (bytes->string/locale #"\316\273")) ; only, so...
bytes->string/locale: byte string is not a valid encoding
for the current locale: #"\316\273"
> (let ([cvt (bytes-open-converter "cp1253" ; Greek code page
                                   "UTF-8")]
        [dest (make-bytes 2)])
    (bytes-convert cvt #"\353" 0 1 dest)
    (bytes-close-converter cvt)
    (bytes->string/utf-8 dest))
"λ"
```

3.6 Symbols

A *symbol* is an atomic value that prints like an identifier preceded with '. An expression that starts with ' and continues with an identifier produces a symbol value.

Examples:
```
> 'a
'a
> (symbol? 'a)
#t
```

For any sequence of characters, exactly one corresponding symbol is *interned*; calling the string->symbol procedure, or reading a syntactic identifier, produces an interned symbol. Since interned symbols can be cheaply compared with eq? (and thus eqv? or equal?), they serves as a convenient values to use for tags and enumerations.

Symbols are case-sensitive. By using a #ci prefix or in other ways, the reader can be made to case-fold character sequences to arrive at a symbol, but the reader preserves case by default.

Examples:
```
> (eq? 'a 'a)
#t
> (eq? 'a (string->symbol "a"))
#t
> (eq? 'a 'b)
#f
> (eq? 'a 'A)
#f
> #ci'A
'a
```

Any string (i.e., any character sequence) can be supplied to string->symbol to obtain the corresponding symbol. For reader input, any character can appear directly in an identifier, except for whitespace and the following special characters:

```
( ) [ ] { } " , ' ` ; # | \
```

Actually, # is disallowed only at the beginning of a symbol, and then only if not followed by %; otherwise, # is allowed, too. Also, . by itself is not a symbol.

Whitespace or special characters can be included in an identifier by quoting them with | or \. These quoting mechanisms are used in the printed form of identifiers that contain special characters or that might otherwise look like numbers.

Examples:
```
> (string->symbol "one, two")
'|one, two|
```

```
> (string->symbol "6")
'|6|
```

The `write` function prints a symbol without a `'` prefix. The `display` form of a symbol is the same as the corresponding string.

Examples:
```
> (write 'Apple)
Apple
> (display 'Apple)
Apple
> (write '|6|)
|6|
> (display '|6|)
6
```

The `gensym` and `string->uninterned-symbol` procedures generate fresh *uninterned* symbols that are not equal (according to `eq?`) to any previously interned or uninterned symbol. Uninterned symbols are useful as fresh tags that cannot be confused with any other value.

Examples:
```
> (define s (gensym))
> s
'g42
> (eq? s 'g42)
> (eq? 'a (string->uninterned-symbol "a"))
#f
```

3.7   Keywords

A *keyword* value is similar to a symbol (see §3.6 "Symbols"), but its printed form is prefixed with `#:`.

Examples:
```
> (string->keyword "apple")
'#:apple
> '#:apple
'#:apple
> (eq? '#:apple (string->keyword "apple"))
#t
```

More precisely, a keyword is analogous to an identifier; in the same way that an identifier can be quoted to produce a symbol, a keyword can be quoted to produce a value. The same term "keyword" is used in both cases, but we sometimes use *keyword value* to refer

50

more specifically to the result of a quote-keyword expression or of `string->keyword`. An unquoted keyword is not an expression, just as an unquoted identifier does not produce a symbol:

Examples:
```
> not-a-symbol-expression
reference to undefined identifier: not-a-symbol-expression
> #:not-a-keyword-expression
eval:2:0: #%datum: keyword used as an expression in:
#:not-a-keyword-expression
```

Despite their similarities, keywords are used in a different way than identifiers or symbols. Keywords are intended for use (unquoted) as special markers in argument lists and in certain syntactic forms. For run-time flags and enumerations, use symbols instead of keywords. The example below illustrates the distinct roles of keywords and symbols.

Examples:
```
> (define dir (find-system-path 'temp-dir)) ; not '#:temp-dir
> (with-output-to-file (build-path dir "stuff.txt")
    (lambda () (printf "example\n"))
    ; optional #:mode argument can be 'text or 'binary
    #:mode 'text
    ; optional #:exists argument can be 'replace, 'truncate, ...
    #:exists 'replace)
```

3.8 Pairs and Lists

A *pair* joins two arbitrary values. The `cons` procedure constructs pairs, and the `car` and `cdr` procedures extract the first and second elements of the pair, respectively. The `pair?` predicate recognizes pairs.

Some pairs print by wrapping parentheses around the printed forms of the two pair elements, putting a ' at the beginning and a . between the elements.

Examples:
```
> (cons 1 2)
'(1 . 2)
> (cons (cons 1 2) 3)
'((1 . 2) . 3)
> (car (cons 1 2))
1
> (cdr (cons 1 2))
2
> (pair? (cons 1 2))
#t
```

51

A *list* is a combination of pairs that creates a linked list. More precisely, a list is either the empty list `null`, or it is a pair whose first element is a list element and whose second element is a list. The `list?` predicate recognizes lists. The `null?` predicate recognizes the empty list.

A list normally prints as a ' followed by a pair of parentheses wrapped around the list elements.

Examples:
```
> null
'()
> (cons 0 (cons 1 (cons 2 null)))
'(0 1 2)
> (list? null)
#t
> (list? (cons 1 (cons 2 null)))
#t
> (list? (cons 1 2))
#f
```

A list or pair prints using `list` or `cons` when one of its elements cannot be written as a quoted value. For example, a value constructed with `srcloc` cannot be written using `quote`, and it prints using `srcloc`:

```
> (srcloc "file.rkt" 1 0 1 (+ 4 4))
(srcloc "file.rkt" 1 0 1 8)
> (list 'here (srcloc "file.rkt" 1 0 1 8) 'there)
(list  'here (srcloc "file.rkt" 1 0 1 8) 'there)
> (cons 1 (srcloc "file.rkt" 1 0 1 8))
(cons  1 (srcloc "file.rkt" 1 0 1 8))
> (cons 1 (cons 2 (srcloc "file.rkt" 1 0 1 8)))
(list*  1 2 (srcloc "file.rkt" 1 0 1 8))
```

See also `list*`.

As shown in the last example, `list*` is used to abbreviate a series of `cons`es that cannot be abbreviated using `list`.

The `write` and `display` functions print a pair or list without a leading ', `cons`, `list`, or `list*`. There is no difference between `write` and `display` for a pair or list, except as they apply to elements of the list:

Examples:
```
> (write (cons 1 2))
(1 . 2)
> (display (cons 1 2))
(1 . 2)
> (write null)
()
```

```
> (display null)
()
> (write (list 1 2 "3"))
(1 2 "3")
> (display (list 1 2 "3"))
(1 2 3)
```

Among the most important predefined procedures on lists are those that iterate through the list's elements:

```
> (map (lambda (i) (/ 1 i))
       '(1 2 3))
'(1 1/2 1/3)
> (andmap (lambda (i) (i . < . 3))
          '(1 2 3))
#f
> (ormap (lambda (i) (i . < . 3))
         '(1 2 3))
#t
> (filter (lambda (i) (i . < . 3))
          '(1 2 3))
'(1 2)
> (foldl (lambda (v i) (+ v i))
         10
         '(1 2 3))
16
> (for-each (lambda (i) (display i))
            '(1 2 3))
123
> (member "Keys"
          '("Florida" "Keys" "U.S.A."))
'("Keys" "U.S.A.")
> (assoc 'where
         '((when "3:30") (where "Florida") (who "Mickey")))
'(where "Florida")
```

Pairs are immutable (contrary to Lisp tradition), and `pair?` and `list?` recognize immutable pairs and lists, only. The `mcons` procedure creates a *mutable pair*, which works with `set-mcar!` and `set-mcdr!`, as well as `mcar` and `mcdr`. A mutable pair prints using `mcons`, while `write` and `display` print mutable pairs with `{` and `}`:

Examples:
```
> (define p (mcons 1 2))
> p
(mcons  1 2)
> (pair? p)
```

```
#f
> (mpair? p)
#t
> (set-mcar! p 0)
> p
(mcons  0 2)
> (write p)
{0 . 2}
```

3.9   Vectors

A *vector* is a fixed-length array of arbitrary values. Unlike a list, a vector supports constant-time access and update of its elements.

A vector prints similar to a list—as a parenthesized sequence of its elements—but a vector is prefixed with # after ', or it uses `vector` if one of its elements cannot be expressed with `quote`.

For a vector as an expression, an optional length can be supplied. Also, a vector as an expression implicitly `quotes` the forms for its content, which means that identifiers and parenthesized forms in a vector constant represent symbols and lists.

Examples:
```
> #("a" "b" "c")
'#("a" "b" "c")
> #(name (that tune))
'#(name (that tune))
> (vector-ref #("a" "b" "c") 1)
"b"
> (vector-ref #(name (that tune)) 1)
'(that tune)
```

Like strings, a vector is either mutable or immutable, and vectors written directly as expressions are immutable.

Vector can be converted to lists and vice versa via `list->vector` and `vector->list`; such conversions are particularly useful in combination with predefined procedures on lists. When allocating extra lists seems too expensive, consider using looping forms like `for/fold`, which recognize vectors as well as lists.

Example:
```
> (list->vector (map string-titlecase
                     (vector->list #("three" "blind" "mice"))))
'#("Three" "Blind" "Mice")
```

54

3.10 Hash Tables

A *hash table* implements a mapping from keys to values, where both keys and values can be arbitrary Scheme values, and access and update to the table are normally constant-time operations. Keys are compared using `equal?`, `eqv?`, or `eq?`, depending on whether the hash table is created with `make-hash`, `make-hasheqv`, or `make-hasheq`.

Examples:
```
> (define ht (make-hash))
> (hash-set! ht "apple" '(red round))
> (hash-set! ht "banana" '(yellow long))
> (hash-ref ht "apple")
'(red round)
> (hash-ref ht "coconut")
hash-ref: no value found for key: "coconut"
> (hash-ref ht "coconut" "not there")
"not there"
```

The `hash`, `hasheqv`, and `hasheq` functions create immutable hash tables from an initial set of keys and values, which each value is provided as an argument after its key. Immutable hash tables can be extended with `hash-set`, which produces a new immutable hash table in constant time.

Examples:
```
> (define ht (hash "apple" 'red "banana" 'yellow))
> (hash-ref ht "apple")
'red
> (define ht2 (hash-set ht "coconut" 'brown))
> (hash-ref ht "coconut")
hash-ref: no value found for key: "coconut"
> (hash-ref ht2 "coconut")
'brown
```

A literal immutable hash table can be written as an expression by using `#hash` (for an `equal?`-based table), `#hasheqv` (for an `eqv?`-based table), or `#hasheq` (for an `eq?`-based table). A parenthesized sequence must immediately follow `#hash`, `#hasheq`, or `#hasheqv`, where each element is a sequence is a dotted key–value pair. The `#hash`, etc. forms implicitly `quote` their key and value sub-forms.

Examples:
```
> (define ht #hash(("apple" . red)
                   ("banana" . yellow)))
> (hash-ref ht "apple")
'red
```

§12.6.11 "Reading Hash Tables" in **Reference**: *Racket* documents the fine points of the syntax of hash table literals.

55

Both mutable and immutable hash tables print like immutable hash tables, using a quoted `#hash`, `#hasheqv`, or `#hasheq` form if all keys and values can be expressed with `quote` or using `hash`, `hasheq`, or `hasheqv` otherwise:

Examples:
```
> #hash(("apple" . red)
        ("banana" . yellow))
'#hash(("apple" . red) ("banana" . yellow))
> (hash 1 (srcloc "file.rkt" 1 0 1 (+ 4 4)))
(hash 1 (srcloc "file.rkt" 1 0 1 8))
```

A mutable hash table can optionally retain its keys *weakly*, so each mapping is retained only so long as the key is retained elsewhere.

Examples:
```
> (define ht (make-weak-hasheq))
> (hash-set! ht (gensym) "can you see me?")
> (collect-garbage)
> (hash-count ht)
0
```

Beware that even a weak hash table retains its values strongly, as long as the corresponding key is accessible. This creates a catch-22 dependency when a value refers back to its key, so that the mapping is retained permanently. To break the cycle, map the key to an ephemeron that pairs the value with its key (in addition to the implicit pairing of the hash table).

Examples:
```
> (define ht (make-weak-hasheq))
> (let ([g (gensym)])
    (hash-set! ht g (list g)))
> (collect-garbage)
> (hash-count ht)
1

> (define ht (make-weak-hasheq))
> (let ([g (gensym)])
    (hash-set! ht g (make-ephemeron g (list g))))
> (collect-garbage)
> (hash-count ht)
0
```

3.11   Boxes

A *box* is like a single-element vector. It can print as a quoted `#&` followed by the printed form of the boxed value. A `#&` form can also be used as an expression, but since the resulting box

56

is constant, it has practically no use.

Examples:
```
> (define b (box "apple"))
> b
'#&"apple"
> (unbox b)
"apple"
> (set-box! b '(banana boat))
> b
'#&(banana boat)
```

3.12   Void and Undefined

Some procedures or expression forms have no need for a result value. For example, the
`display` procedure is called only for the side-effect of writing output. In such cases the
result value is normally a special constant that prints as `#<void>`. When the result of an
expression is simply `#<void>`, the REPL does not print anything.

The `void` procedure takes any number of arguments and returns `#<void>`. (That is, the
identifier `void` is bound to a procedure that returns `#<void>`, instead of being bound directly
to `#<void>`.)

Examples:
```
> (void)
> (void 1 2 3)
> (list (void))
'(#<void>)
```

A constant that prints as `#<undefined>` is used as the result of a reference to a local binding
when the binding is not yet initialized. Such early references are not possible for bindings
that correspond to procedure arguments, `let` bindings, or `let*` bindings; early reference
requires a recursive binding context, such as `letrec` or local `define`s in a procedure body.
Also, early references to top-level and module-level bindings raise an exception, instead of
producing `#<undefined>`. For these reasons, `#<undefined>` rarely appears.

```
(define (strange)
  (define x x)
  x)

> (strange)
#<undefined>
```

57

4 Expressions and Definitions

The §2 "Racket Essentials" chapter introduced some of Racket's syntactic forms: definitions, procedure applications, conditionals, and so on. This section provides more details on those forms, plus a few additional basic forms.

4.1 Notation

This chapter (and the rest of the documentation) uses a slightly different notation than the character-based grammars of the §2 "Racket Essentials" chapter. The grammar for a use of a syntactic form `something` is shown like this:

```
(something [id ...+] an-expr ...)
```

The italicized meta-variables in this specification, such as `id` and `an-expr`, use the syntax of Racket identifiers, so `an-expr` is one meta-variable. A naming convention implicitly defines the meaning of many meta-variables:

- A meta-variable that ends in `id` stands for an identifier, such as `x` or `my-favorite-martian`.

- A meta-identifier that ends in `keyword` stands for a keyword, such as `#:tag`.

- A meta-identifier that ends with `expr` stands for any sub-form, and it will be parsed as an expression.

- A meta-identifier that ends with `body` stands for any sub-form; it will be parsed as either a local definition or an expression. A `body` can parse as a definition only if it is not preceded by any expression, and the last `body` must be an expression; see also §4.5.4 "Internal Definitions".

Square brackets in the grammar indicate a parenthesized sequence of forms, where square brackets are normally used (by convention). That is, square brackets *do not* mean optional parts of the syntactic form.

A `...` indicates zero or more repetitions of the preceding form, and `...+` indicates one or more repetitions of the preceding datum. Otherwise, non-italicized identifiers stand for themselves.

Based on the above grammar, then, here are a few conforming uses of `something`:

```
(something [x])
```

```
(something [x] (+ 1 2))
(something [x my-favorite-martian x] (+ 1 2) #f)
```

Some syntactic-form specifications refer to meta-variables that are not implicitly defined and not previously defined. Such meta-variables are defined after the main form, using a BNF-like format for alternatives:

```
(something-else [thing ...+] an-expr ...)

thing = thing-id
      | thing-keyword
```

The above example says that, within a `something-else` form, a *thing* is either an identifier or a keyword.

4.2 Identifiers and Binding

The context of an expression determines the meaning of identifiers that appear in the expression. In particular, starting a module with the language `racket`, as in

```
#lang racket
```

means that, within the module, the identifiers described in this guide start with the meaning described here: `cons` refers to the function that creates a pair, `car` refers to the function that extracts the first element of a pair, and so on.

Forms like `define`, `lambda`, and `let` associate a meaning with one or more identifiers; that is, they *bind* identifiers. The part of the program for which the binding applies is the *scope* of the binding. The set of bindings in effect for a given expression is the expression's *environment*.

For example, in

```
#lang racket

(define f
  (lambda (x)
    (let ([y 5])
      (+ x y))))

(f 10)
```

the `define` is a binding of `f`, the `lambda` has a binding for `x`, and the `let` has a binding for `y`. The scope of the binding for `f` is the entire module; the scope of the `x` binding is `(let`

`([y 5]) (+ x y))`; and the scope of the `y` binding is just `(+ x y)`. The environment of `(+ x y)` includes bindings for `y`, `x`, and `f`, as well as everything in `racket`.

A module-level `define` can bind only identifiers that are not already bound within the module. For example, `(define cons 1)` is a syntax error in a `racket` module, since `cons` is provided by `racket`. A local `define` or other binding forms, however, can give a new local binding for an identifier that already has a binding; such a binding *shadows* the existing binding.

Examples:
```
(define f
  (lambda (append)
    (define cons (append "ugly" "confusing"))
    (let ([append 'this-was])
      (list append cons))))
> (f list)
'(this-was ("ugly" "confusing"))
```

Even identifiers like `define` and `lambda` get their meanings from bindings, though they have *transformer* bindings (which means that they indicate syntactic forms) instead of value bindings. Since `define` has a transformer binding, the identifier `define` cannot be used by itself to get a value. However, the normal binding for `define` can be shadowed.

Examples:
```
> define
eval:1:0: define: bad syntax in: define
> (let ([define 5]) define)
5
```

Shadowing standard bindings in this way is rarely a good idea, but the possibility is an inherent part of Racket's flexibility.

4.3   Function Calls (Procedure Applications)

An expression of the form

```
(proc-expr arg-expr ...)
```

---

is a function call—also known as a *procedure application*—when *proc-expr* is not an identifier that is bound as a syntax transformer (such as `if` or `define`).

4.3.1 Evaluation Order and Arity

A function call is evaluated by first evaluating the *proc-expr* and all *arg-expr*s in order
(left to right). Then, if *proc-expr* produces a function that accepts as many arguments as
supplied *arg-expr*s, the function is called. Otherwise, an exception is raised.

Examples:
```
> (cons 1 null)
'(1)
> (+ 1 2 3)
6
> (cons 1 2 3)
cons: expects 2 arguments, given 3: 1 2 3
> (1 2 3)
procedure application: expected procedure, given: 1;
arguments were: 2 3
```

Some functions, such as `cons`, accept a fixed number of arguments. Some functions, such
as `+` or `list`, accept any number of arguments. Some functions accept a range of argument
counts; for example `substring` accepts either two or three arguments. A function's *arity* is
the number of arguments that it accepts.

4.3.2 Keyword Arguments

Some functions accept *keyword arguments* in addition to by-position arguments. For that
case, an `arg` can be an `arg-keyword arg-expr` sequence instead of just a `arg-expr`:

```
(proc-expr arg ...)

arg = arg-expr
    | arg-keyword arg-expr
```

For example,

```
(go "super.ss" #:mode 'fast)
```

calls the function bound to `go` with `"super.ss"` as a by-position argument, and with `'fast`
as an argument associated with the `#:mode` keyword. A keyword is implicitly paired with
the expression that follows it.

Since a keyword by itself is not an expression, then

```
(go "super.ss" #:mode #:fast)
```

is a syntax error. The `#:mode` keyword must be followed by an expression to produce an argument value, and `#:fast` is not an expression.

The order of keyword `arg`s determines the order in which `arg-expr`s are evaluated, but a function accepts keyword arguments independent of their position in the argument list. The above call to `go` can be equivalently written

```
(go #:mode 'fast "super.ss")
```

4.3.3   The `apply` Function

The syntax for function calls supports any number of arguments, but a specific call always specifies a fixed number of arguments. As a result, a function that takes a list of arguments cannot directly apply a function like `+` to all of the items in the list:

```
(define (avg lst) ; doesn't work...
  (/ (+ lst) (length lst)))

> (avg '(1 2 3))
+: expects argument of type <number>; given '(1 2 3)

(define (avg lst) ; doesn't always work...
 (/ (+ (list-ref lst 0) (list-ref lst 1) (list-ref lst 2))
    (length lst)))

> (avg '(1 2 3))
2
> (avg '(1 2))
list-ref: index 2 too large for list: '(1 2)
```

The `apply` function offers a way around this restriction. It takes a function and a *list* arguments, and it applies the function to the arguments:

```
(define (avg lst)
  (/ (apply + lst) (length lst)))

> (avg '(1 2 3))
2
> (avg '(1 2))
3/2
> (avg '(1 2 3 4))
5/2
```

As a convenience, the `apply` function accepts additional arguments between the function and the list. The additional arguments are effectively `cons`ed onto the argument list:

62

```
(define (anti-sum lst)
  (apply - 0 lst))

> (anti-sum '(1 2 3))
-6
```

The `apply` function supports only by-position arguments. To apply a function with keyword arguments, use the `keyword-apply` function, which accepts a function to apply and three lists. The first two lists are in parallel, where the first list contains keywords (sorted by `keyword<`), and the second list contains a corresponding argument for each keyword. The third list contains by-position function arguments, as for `apply`.

```
(keyword-apply go
               '(#:mode)
               '(fast)
               '("super.ss"))
```

4.4   Functions (Procedures): `lambda`

A `lambda` expression creates a function. In the simplest case, a `lambda` expression has the form

---

```
(lambda (arg-id ...)
  body ...+)
```

A `lambda` form with *n* `arg-id`s accepts *n* arguments:

```
> ((lambda (x) x)
   1)
1
> ((lambda (x y) (+ x y))
   1 2)
3
> ((lambda (x y) (+ x y))
   1)
#<procedure>: expects 2 arguments, given 1: 1
```

4.4.1   Declaring a Rest Argument

A `lambda` expression can also have the form

```
(lambda rest-id
  body ...+)
```

That is, a lambda expression can have a single *rest-id* that is not surrounded by parentheses. The resulting function accepts any number of arguments, and the arguments are put into a list bound to *rest-id*.

Examples:
```
> ((lambda x x)
   1 2 3)
'(1 2 3)
> ((lambda x x))
'()
> ((lambda x (car x))
   1 2 3)
1
```

Functions with a *rest-id* often use apply to call another function that accepts any number of arguments.

Examples:
```
(define max-mag
  (lambda nums
    (apply max (map magnitude nums))))
> (max 1 -2 0)
1
> (max-mag 1 -2 0)
2
```

The lambda form also supports required arguments combined with a *rest-id*:

```
(lambda (arg-id ...+ . rest-id)
  body ...+)
```

The result of this form is a function that requires at least as many arguments as *arg-id*s, and also accepts any number of additional arguments.

Examples:
```
(define max-mag
  (lambda (num . nums)
    (apply max (map magnitude (cons num nums)))))
> (max-mag 1 -2 0)
2
> (max-mag)
```

A `rest-id` variable is sometimes called a *rest argument*, because it accepts the "rest" of the function arguments.

4.4.2 Declaring Optional Arguments

Instead of just an identifier, an argument (other than a rest argument) in a `lambda` form can be specified with an identifier and a default value:

```
(lambda gen-formals
  body ...+)

gen-formals = (arg ...)
            | rest-id
            | (arg ...+ . rest-id)

        arg = arg-id
            | [arg-id default-expr]
```

A argument of the form `[arg-id default-expr]` is optional. When the argument is not supplied in an application, `default-expr` produces the default value. The `default-expr` can refer to any preceding `arg-id`, and every following `arg-id` must have a default as well.

Examples:
```
(define greet
  (lambda (given [surname "Smith"])
    (string-append "Hello, " given " " surname)))
> (greet "John")
"Hello, John Smith"
> (greet "John" "Doe")
"Hello, John Doe"

(define greet
  (lambda (given [surname (if (equal? given "John")
                              "Doe"
                              "Smith")])
    (string-append "Hello, " given " " surname)))

> (greet "John")
"Hello, John Doe"
> (greet "Adam")
"Hello, Adam Smith"
```

4.4.3 Declaring Keyword Arguments

A `lambda` form can declare an argument to be passed by keyword, instead of position. Keyword arguments can be mixed with by-position arguments, and default-value expressions can be supplied for either kind of argument:

```
(lambda gen-formals
  body ...+)

gen-formals = (arg ...)
            | rest-id
            | (arg ...+ . rest-id)

        arg = arg-id
            | [arg-id default-expr]
            | arg-keyword arg-id
            | arg-keyword [arg-id default-expr]
```

An argument specified as `arg-keyword arg-id` is supplied by an application using the same `arg-keyword`. The position of the keyword–identifier pair in the argument list does not matter for matching with arguments in an application, because it will be matched to an argument value by keyword instead of by position.

```
(define greet
  (lambda (given #:last surname)
    (string-append "Hello, " given " " surname)))

> (greet "John" #:last "Smith")
"Hello, John Smith"
> (greet #:last "Doe" "John")
"Hello, John Doe"
```

An `arg-keyword [arg-id default-expr]` argument specifies a keyword-based argument with a default value.

Examples:
```
(define greet
  (lambda (#:hi [hi "Hello"] given #:last [surname "Smith"])
    (string-append hi ", " given " " surname)))
> (greet "John")
"Hello, John Smith"
> (greet "Karl" #:last "Marx")
"Hello, Karl Marx"
> (greet "John" #:hi "Howdy")
```

```
"Howdy, John Smith"
> (greet "Karl" #:last "Marx" #:hi "Guten Tag")
"Guten Tag, Karl Marx"
```

The `lambda` form does not directly support the creation of a function that accepts "rest" keywords. To construct a function that accepts all keyword arguments, use `make-keyword-procedure`. The function supplied to `make-keyword-procedure` receives keyword arguments through parallel lists in the first two (by-position) arguments, and then all by-position arguments from an application as the remaining by-position arguments.

Examples:
```
(define (trace-wrap f)
  (make-keyword-procedure
   (lambda (kws kw-args . rest)
     (printf "Called with ~s ~s ~s\n" kws kw-args rest)
     (keyword-apply f kws kw-args rest))))
> ((trace-wrap greet) "John" #:hi "Howdy")
Called with (#:hi) ("Howdy") ("John")
"Howdy, John Smith"
```

4.4.4 Arity-Sensitive Functions: `case-lambda`

The `case-lambda` form creates a function that can have completely different behaviors depending on the number of arguments that are supplied. A case-lambda expression has the form

```
(case-lambda
  [formals body ...+]
  ...)

formals = (arg-id ...)
        | rest-id
        | (arg-id ...+ . rest-id)
```

where each `[formals body ...+]` is analogous to `(lambda formals body ...+)`. Applying a function produced by `case-lambda` is like applying a `lambda` for the first case that matches the number of given arguments.

Examples:
```
(define greet
  (case-lambda
    [(name) (string-append "Hello, " name)]
    [(given surname) (string-append "Hello, " given " " surname)]))
```

67

```
> (greet "John")
"Hello, John"
> (greet "John" "Smith")
"Hello, John Smith"
> (greet)
procedure greet: no clause matching 0 arguments
```

A `case-lambda` function cannot directly support optional or keyword arguments.

4.5   Definitions: `define`

A basic definition has the form

```
(define id expr)
```

in which case `id` is bound to the result of `expr`.

Examples:
```
  (define salutation (list-ref '("Hi" "Hello") (random 2)))
  > salutation
  "Hello"
```

4.5.1   Function Shorthand

The `define` form also supports a shorthand for function definitions:

```
(define (id arg ...) body ...+)
```

which is a shorthand for

```
  (define id (lambda (arg ...) body ...+))
```

Examples:
```
  (define (greet name)
    (string-append salutation ", " name))
  > (greet "John")
  "Hello, John"

  (define (greet first [surname "Smith"] #:hi [hi salutation])
    (string-append hi ", " first " " surname))
```

```
> (greet "John")
"Hello, John Smith"
> (greet "John" #:hi "Hey")
"Hey, John Smith"
> (greet "John" "Doe")
"Hello, John Doe"
```

The function shorthand via `define` also supports a "rest" argument (i.e., a final argument to collect extra arguments in a list):

```
(define (id arg ... . rest-id) body ...+)
```

which is a shorthand

```
(define id (lambda (arg ... . rest-id) body ...+))
```

Examples:
```
(define (avg . l)
  (/ (apply + l) (length l)))
> (avg 1 2 3)
2
```

4.5.2   Curried Function Shorthand

Consider the following `make-add-suffix` function that takes a string and returns another function that takes a string:

```
(define make-add-suffix
  (lambda (s2)
    (lambda (s) (string-append s s2))))
```

Although it's not common, result of `make-add-suffix` could be called directly, like this:

```
> ((make-add-suffix "!") "hello")
"hello!"
```

In a sense, `make-add-suffix` is a function takes two arguments, but it takes them one at a time. A function that takes some of its arguments and returns a function to consume more is sometimes called a *curried function*.

Using the function-shorthand form of `define`, `make-add-suffix` can be written equivalently as

```
(define (make-add-suffix s2)
  (lambda (s) (string-append s s2)))
```

This shorthand reflects the shape of the function call (make-add-suffix "!"). The de-fine form further supports a shorthand for defining curried functions that reflects nested function calls:

```
(define ((make-add-suffix s2) s)
  (string-append s s2))

> ((make-add-suffix "!") "hello")
"hello!"

(define louder (make-add-suffix "!"))
(define less-sure (make-add-suffix "?"))

> (less-sure "really")
"really?"
> (louder "really")
"really!"
```

The full syntax of the function shorthand for define is as follows:

```
(define (head args) body ...+)

head = id
     | (head args)

args = arg ...
     | arg ... . rest-id
```

The expansion of this shorthand has one nested lambda form for each *head* in the definition, where the innermost *head* corresponds to the outermost lambda.

4.5.3 Multiple Values and define-values

A Racket expression normally produces a single result, but some expressions can produce multiple results. For example, quotient and remainder each produce a single value, but quotient/remainder produces the same two values at once:

```
> (quotient 13 3)
4
> (remainder 13 3)
1
```

```
> (quotient/remainder 13 3)
4
1
```

As shown above, the REPL prints each result value on its own line.

Multiple-valued functions can be implemented in terms of the `values` function, which takes any number of values and returns them as the results:

```
> (values 1 2 3)
1
2
3

(define (split-name name)
  (let ([parts (regexp-split " " name)])
    (if (= (length parts) 2)
        (values (list-ref parts 0) (list-ref parts 1))
        (error "not a <first> <last> name"))))

> (split-name "Adam Smith")
"Adam"
"Smith"
```

The `define-values` form binds multiple identifiers at once to multiple results produced from a single expression:

```
(define-values (id ...) expr)
```

The number of results produced by the `expr` must match the number of `id`s.

Examples:
```
(define-values (given surname) (split-name "Adam Smith"))
> given
"Adam"
> surname
"Smith"
```

A `define` form (that is not a function shorthand) is equivalent to a `define-values` form with a single `id`.

71

4.5.4 Internal Definitions

When the grammar for a syntactic form specifies *body*, then the corresponding form can be either a definition or an expression. A definition as a *body* is an *internal definition*.

Expressions and internal definitions in a *body* sequence can be mixed, as long as the last *body* is an expression.

For example, the syntax of `lambda` is

```
(lambda gen-formals
  body ...+)
```

so the following are valid instances of the grammar:

```
(lambda (f)                 ; no definitions
  (printf "running\n")
  (f 0))

(lambda (f)                 ; one definition
  (define (log-it what)
    (printf "~a\n"))
  (log-it "running")
  (f 0)
  (log-it "done"))

(lambda (f n)               ; two definitions
  (define (call n)
    (if (zero? n)
        (log-it "done")
        (begin
          (log-it "running")
          (f 0)
          (call (- n 1)))))
  (define (log-it what)
    (printf "~a\n"))
  (call f n))
```

Internal definitions in a particular *body* sequence are mutually recursive; that is, any definition can refer to any other definition—as long as the reference isn't actually evaluated before its definition takes place. If a definition is referenced too early, the result is a special value `#<undefined>`.

Examples:
```
  (define (weird)
```

```
    (define x x)
    x)
> (weird)
#<undefined>
```

A sequence of internal definitions using just `define` is easily translated to an equivalent `letrec` form (as introduced in the next section). However, other definition forms can appear as a *body*, including `define-values`, `struct` (see §5 "Programmer-Defined Datatypes") or `define-syntax` (see §16 "Macros").

4.6 Local Binding

Although internal `define`s can be used for local binding, Racket provides three forms that give the programmer more control over bindings: `let`, `let*`, and `letrec`.

4.6.1 Parallel Binding: `let`

A `let` form binds a set of identifiers, each to the result of some expression, for use in the `let` body:

```
(let ([id expr] ...) body ...+)
```

The *id*s are bound "in parallel." That is, no *id* is bound in the right-hand side *expr* for any *id*, but all are available in the *body*. The *id*s must be different from each other.

Examples:
```
> (let ([me "Bob"])
    me)
"Bob"
> (let ([me "Bob"]
        [myself "Robert"]
        [I "Bobby"])
    (list me myself I))
'("Bob" "Robert" "Bobby")
> (let ([me "Bob"]
        [me "Robert"])
    me)
```
eval:3:0: let: duplicate identifier at: me in: (let ((me "Bob") (me "Robert")) me)

The fact that an *id*'s *expr* does not see its own binding is often useful for wrappers that must refer back to the old value:

73

```
> (let ([+ (lambda (x y)
             (if (string? x)
                 (string-append x y)
                 (+ x y)))]) ; use original +
    (list (+ 1 2)
          (+ "see" "saw")))
'(3 "seesaw")
```

Occasionally, the parallel nature of `let` bindings is convenient for swapping or rearranging a set of bindings:

```
> (let ([me "Tarzan"]
        [you "Jane"])
    (let ([me you]
          [you me])
      (list me you)))
'("Jane" "Tarzan")
```

The characterization of `let` bindings as "parallel" is not meant to imply concurrent evaluation. The *exprs* are evaluated in order, even though the bindings are delayed until all *exprs* are evaluated.

4.6.2 Sequential Binding: `let*`

The syntax of `let*` is the same as `let`:

```
(let* ([id expr] ...) body ...+)
```

The difference is that each `id` is available for use in later *exprs*, as well as in the *body*. Furthermore, the `id`s need not be distinct, and the most recent binding is the visible one.

Examples:
```
> (let* ([x (list "Borroughs")]
         [y (cons "Rice" x)]
         [z (cons "Edgar" y)])
    (list x y z))
'(("Borroughs") ("Rice" "Borroughs") ("Edgar" "Rice" "Borroughs"))
> (let* ([name (list "Borroughs")]
         [name (cons "Rice" name)]
         [name (cons "Edgar" name)])
    name)
'("Edgar" "Rice" "Borroughs")
```

In other words, a `let*` form is equivalent to nested `let` forms, each with a single binding:

74

```
> (let ([name (list "Borroughs")])
    (let ([name (cons "Rice" name)])
      (let ([name (cons "Edgar" name)])
        name)))
'("Edgar" "Rice" "Borroughs")
```

4.6.3 Recursive Binding: letrec

The syntax of letrec is also the same as let:

```
(letrec ([id expr] ...) body ...+)
```

While let makes its bindings available only in the *body*s, and let* makes its bindings available to any later binding *expr*, letrec makes its bindings available to all other *expr*s—even earlier ones. In other words, letrec bindings are recursive.

The *expr*s in a letrec form are most often lambda forms for recursive and mutually recursive functions:

```
> (letrec ([swing
            (lambda (t)
              (if (eq? (car t) 'tarzan)
                  (cons 'vine
                        (cons 'tarzan (cddr t)))
                  (cons (car t)
                        (swing (cdr t)))))])
    (swing '(vine tarzan vine vine)))
'(vine vine tarzan vine)

> (letrec ([tarzan-in-tree?
            (lambda (name path)
              (or (equal? name "tarzan")
                  (and (directory-exists? path)
                       (tarzan-in-directory? path))))]
           [tarzan-in-directory?
            (lambda (dir)
              (ormap (lambda (elem)
                       (tarzan-in-tree? (path-element->string elem)
                                        (build-path dir elem)))
                     (directory-list dir)))])
    (tarzan-in-tree? "tmp" (find-system-path 'temp-dir)))
#f
```

75

While the *expr*s of a `letrec` form are typically `lambda` expressions, they can be any expression. The expressions are evaluated in order, and after each value is obtained, it is immediately associated with its corresponding *id*. If an *id* is referenced before its value is ready, the result is `#<undefined>`, as just as for internal definitions.

```
> (letrec ([quicksand quicksand])
    quicksand)
#<undefined>
```

4.6.4 Named `let`

A named `let` is an iteration and recursion form. It uses the same syntactic keyword `let` as for local binding, but an identifier after the `let` (instead of an immediate open parenthesis) triggers a different parsing.

```
(let _proc-id ([_arg-id _init-expr] ...)
  _body ...+)
```

A named `let` form is equivalent to

```
(letrec ([proc-id (lambda (arg-id ...)
                    body ...+)])
  (proc-id init-expr ...))
```

That is, a named `let` binds a function identifier that is visible only in the function's body, and it implicitly calls the function with the values of some initial expressions.

Examples:
```
(define (duplicate pos lst)
  (let dup ([i 0]
            [lst lst])
    (cond
     [(= i pos) (cons (car lst) lst)]
     [else (cons (car lst) (dup (+ i 1) (cdr lst)))])))
> (duplicate 1 (list "apple" "cheese burger!" "banana"))
'("apple" "cheese burger!" "cheese burger!" "banana")
```

4.6.5 Multiple Values: `let-values`, `let*-values`, `letrec-values`

In the same way that `define-values` binds multiple results in a definition (see §4.5.3 "Multiple Values and `define-values`"), `let-values`, `let*-values`, and `letrec-values` bind multiple results locally.

§2.9 "Local Binding: `let`, `let*`, `letrec`, ..." in *Reference: Racket* also documents multiple-value binding forms.

76

```
(let-values ([(id ...) expr] ...)
  body ...+)
```

```
(let*-values ([(id ...) expr] ...)
  body ...+)
```

```
(letrec-values ([(id ...) expr] ...)
  body ...+)
```

Each *expr* must produce as many values as corresponding *id*s. The binding rules are the same for the forms without -values forms: the *id*s of let-values are bound only in the *body*s, the *id*s of let*-valuess are bound in *expr*s of later clauses, and the *id*s of letrec-values are bound for all *expr*s.

Example:
```
> (let-values ([(q r) (quotient/remainder 14 3)])
    (list q r))
'(4 2)
```

4.7 Conditionals

Most functions used for branching, such as < and string?, produce either #t or #f. Racket's branching forms, however, treat any value other than #f as true. We say a *true value* to mean any value other than #f.

This convention for "true value" meshes well with protocols where #f can serve as failure or to indicate that an optional value is not supplied. (Beware of overusing this trick, and remember that an exception is usually a better mechanism to report failure.)

For example, the member function serves double duty; it can be used to find the tail of a list that starts with a particular item, or it can be used to simply check whether an item is present in a list:

```
> (member "Groucho" '("Harpo" "Zeppo"))
#f
> (member "Groucho" '("Harpo" "Groucho" "Zeppo"))
'("Groucho" "Zeppo")
> (if (member "Groucho" '("Harpo" "Zeppo"))
      'yep
```

```
      'nope)
'nope
> (if (member "Groucho" '("Harpo" "Groucho" "Zeppo"))
      'yep
      'nope)
'yep
```

4.7.1 Simple Branching: `if`

In an `if` form,

```
(if test-expr then-expr else-expr)
```

the `test-expr` is always evaluated. If it produces any value other than `#f`, then `then-expr` is evaluated. Otherwise, `else-expr` is evaluated.

An `if` form must have both a `then-expr` and an `else-expr`; the latter is not optional. To perform (or skip) side-effects based on a `test-expr`, use `when` or `unless`, which we describe later in §4.8 "Sequencing".

4.7.2 Combining Tests: `and` and `or`

Racket's `and` and `or` are syntactic forms, rather than functions. Unlike a function, the `and` and `or` forms can skip evaluation of later expressions if an earlier one determines the answer.

```
(and expr ...)
```

An `and` form produces `#f` if any of its `expr`s produces `#f`. Otherwise, it produces the value of its last `expr`. As a special case, `(and)` produces `#t`.

```
(or expr ...)
```

The `or` form produces `#f` if all of its `expr`s produce `#f`. Otherwise, it produces the first non-`#f` value from its `expr`s. As a special case, `(or)` produces `#f`.

Examples:
```
> (define (got-milk? lst)
    (and (not (null? lst))
```

```
          (or (eq? 'milk (car lst))
              (got-milk? (cdr lst)))))) ; recurs only if needed
> (got-milk? '(apple banana))
#f
> (got-milk? '(apple milk banana))
#t
```

If evaluation reaches the last *expr* of an and or or form, then the *expr*'s value directly
determines the and or or result. Therefore, the last *expr* is in tail position, which means
that the above `got-milk?` function runs in constant space.

§2.3.3 "Tail
Recursion"
introduces tail calls
and tail positions.

4.7.3 Chaining Tests: cond

The `cond` form chains a series of tests to select a result expression. To a first approximation,
the syntax of `cond` is as follows:

§2.12
"Conditionals: `if`,
`cond`, `and`, and `or`"
in ***Reference:
Racket*** also
documents `cond`.

```
(cond [test-expr expr ...+]
      ...)
```

Each *test-expr* is evaluated in order. If it produces `#f`, the corresponding *expr*s are
ignored, and evaluation proceeds to the next *test-expr*. As soon as a *test-expr* produces
a true value, its *text-expr*s are evaluated to produce the result for the `cond` form, and no
further *test-expr*s are evaluated.

The last *test-expr* in a `cond` can be replaced by `else`. In terms of evaluation, `else` serves
as a synonym for `#t`, but it clarifies that the last clause is meant to catch all remaining cases.
If `else` is not used, then it is possible that no *test-expr*s produce a true value; in that case,
the result of the `cond` expression is `#<void>`.

Examples:
```
> (cond
    [(= 2 3) (error "wrong!")]
    [(= 2 2) 'ok])
'ok
> (cond
    [(= 2 3) (error "wrong!")])
> (cond
    [(= 2 3) (error "wrong!")]
    [else 'ok])
'ok

(define (got-milk? lst)
  (cond
```

```
      [(null? lst) #f]
      [(eq? 'milk (car lst)) #t]
      [else (got-milk? (cdr lst))]]))

> (got-milk? '(apple banana))
#f
> (got-milk? '(apple milk banana))
#t
```

The full syntax of cond includes two more kinds of clauses:

```
(cond cond-clause ...)

cond-clause = [test-expr then-expr ...+]
            | [else then-expr ...+]
            | [test-expr => proc-expr]
            | [test-expr]
```

The => variant captures the true result of its *test-expr* and passes it to the result of the *proc-expr*, which must be a function of one argument.

Examples:
```
> (define (after-groucho lst)
    (cond
      [(member "Groucho" lst) => cdr]
      [else (error "not there")]))
> (after-groucho '("Harpo" "Groucho" "Zeppo"))
'("Zeppo")
> (after-groucho '("Harpo" "Zeppo"))
not there
```

A clause that includes only a *test-expr* is rarely used. It captures the true result of the *test-expr*, and simply returns the result for the whole cond expression.

4.8   Sequencing

Racket programmers prefer to write programs with as few side-effects as possible, since purely functional code is more easily tested and composed into larger programs. Interaction with the external environment, however, requires sequencing, such as when writing to a display, opening a graphical window, or manipulating a file on disk.

4.8.1 Effects Before: `begin`

A `begin` expression sequences expressions:

```
(begin expr ...+)
```

The *expr*s are evaluated in order, and the result of all but the last *expr* is ignored. The result from the last *expr* is the result of the `begin` form, and it is in tail position with respect to the `begin` form.

Examples:
```
(define (print-triangle height)
  (if (zero? height)
      (void)
      (begin
        (display (make-string height #\*))
        (newline)
        (print-triangle (sub1 height)))))
> (print-triangle 4)
****
***
**
*
```

Many forms, such as `lambda` or `cond` support a sequence of expressions even without a `begin`. Such positions are sometimes said to have an *implicit begin*.

Examples:
```
(define (print-triangle height)
  (cond
    [(positive? height)
     (display (make-string height #\*))
     (newline)
     (print-triangle (sub1 height))]))
> (print-triangle 4)
****
***
**
*
```

The `begin` form is special at the top level, at module level, or as a `body` after only internal definitions. In those positions, instead of forming an expression, the content of `begin` is spliced into the surrounding context.

Example:

```
> (let ([curly 0])
    (begin
     (define moe (+ 1 curly))
     (define larry (+ 1 moe)))
    (list larry curly moe))
'(2 0 1)
```

This splicing behavior is mainly useful for macros, as we discuss later in §16 "Macros".

4.8.2   Effects After: `begin0`

A `begin0` expression has the same syntax as a `begin` expression:

```
(begin0 expr ...+)
```

The difference is that `begin0` returns the result of the first `expr`, instead of the result of the last `expr`. The `begin0` form is useful for implementing side-effects that happen after a computation, especially in the case where the computation produces an unknown number of results.

Examples:
```
  (define (log-times thunk)
    (printf "Start: ~s\n" (current-inexact-milliseconds))
    (begin0
      (thunk)
      (printf "End..: ~s\n" (current-inexact-milliseconds))))
> (log-times (lambda () (sleep 0.1) 0))
Start: 1297720981746.851
End..: 1297720981847.066
0
> (log-times (lambda () (values 1 2)))
Start: 1297720981847.594
End..: 1297720981847.678
1
2
```

4.8.3   Effects If...: `when` and `unless`

The `when` form combines an `if`-style conditional with sequencing for the "then" clause and no "else" clause:

```
(when test-expr then-expr ...)
```

If `test-expr` produces a true value, then all of the `then-expr`s are evaluated. Otherwise, no `then-expr`s are evaluated. The result is `#<void>` in any case.

The `unless` form is similar:

```
(unless test-expr then-expr ...)
```

The difference is that the `test-expr` result is inverted: the `then-expr`s are evaluated only if the `test-expr` result is `#f`.

Examples:
```
  (define (enumerate lst)
   (if (null? (cdr lst))
       (printf "~a.\n" (car lst))
       (begin
         (printf "~a, " (car lst))
         (when (null? (cdr (cdr lst)))
           (printf "and "))
         (enumerate (cdr lst)))))
> (enumerate '("Larry" "Curly" "Moe"))
Larry, Curly, and Moe.

  (define (print-triangle height)
    (unless (zero? height)
      (display (make-string height #\*))
      (newline)
      (print-triangle (sub1 height))))

> (print-triangle 4)
****
***
**
*
```

4.9   Assignment: `set!`

Assign to a variable using `set!`:

```
(set! id expr)
```

A `set!` expression evaluates *expr* and changes *id* (which must be bound in the enclosing environment) to the resulting value. The result of the `set!` expression itself is `#<void>`.

Examples:
```
(define greeted null)
(define (greet name)
  (set! greeted (cons name greeted))
  (string-append "Hello, " name))
> (greet "Athos")
"Hello, Athos"
> (greet "Porthos")
"Hello, Porthos"
> (greet "Aramis")
"Hello, Aramis"
> greeted
'("Aramis" "Porthos" "Athos")

(define (make-running-total)
  (let ([n 0])
    (lambda ()
      (set! n (+ n 1))
      n)))
(define win (make-running-total))
(define lose (make-running-total))

> (win)
1
> (win)
2
> (lose)
1
> (win)
3
```

4.9.1   Guidelines for Using Assignment

Although using `set!` is sometimes appropriate, Racket style generally discourages the use of `set!`. The following guidelines may help explain when using `set!` is appropriate.

- As in any modern language, assigning to shared identifier is no substitute for passing an argument to a procedure or getting its result.

  Really awful example:
  ```
  (define name "unknown")
  (define result "unknown")
  ```

```
(define (greet)
  (set! result (string-append "Hello, " name)))

> (set! name "John")
> (greet)
> result
"Hello, John"
```
Ok example:
```
(define (greet name)
  (string-append "Hello, " name))

> (greet "John")
"Hello, John"
> (greet "Anna")
"Hello, Anna"
```

- A sequence of assignments to a local variable is far inferior to nested bindings.

  Bad example:
  ```
  > (let ([tree 0])
      (set! tree (list tree 1 tree))
      (set! tree (list tree 2 tree))
      (set! tree (list tree 3 tree))
      tree)
  '(((0 1 0) 2 (0 1 0)) 3 ((0 1 0) 2 (0 1 0)))
  ```
  Ok example:
  ```
  > (let* ([tree 0]
           [tree (list tree 1 tree)]
           [tree (list tree 2 tree)]
           [tree (list tree 3 tree)])
      tree)
  '(((0 1 0) 2 (0 1 0)) 3 ((0 1 0) 2 (0 1 0)))
  ```

- Using assignment to accumulate results from an iteration is bad style. Accumulating through a loop argument is better.

  Somewhat bad example:
  ```
  (define (sum lst)
    (let ([s 0])
      (for-each (lambda (i) (set! s (+ i s)))
                lst)
      s))

  > (sum '(1 2 3))
  6
  ```
  Ok example:
```

```
(define (sum lst)
  (let loop ([lst lst] [s 0])
    (if (null? lst)
        s
        (loop (cdr lst) (+ s (car lst))))))

> (sum '(1 2 3))
6
```

Better (use an existing function) example:
```
(define (sum lst)
  (apply + lst))

> (sum '(1 2 3))
6
```

Good (a general approach) example:
```
(define (sum lst)
  (for/fold ([s 0])
            ([i (in-list lst)])
    (+ s i)))

> (sum '(1 2 3))
6
```

- For cases where stateful objects are necessary or appropriate, then implementing the object's state with set! is fine.

Ok example:
```
(define next-number!
  (let ([n 0])
    (lambda ()
      (set! n (add1 n))
      n)))

> (next-number!)
1
> (next-number!)
2
> (next-number!)
3
```

All else being equal, a program that uses no assignments or mutation is always preferable to one that uses assignments or mutation. While side effects are to be avoided, however, they should be used if the resulting code is significantly more readable or if it implements a significantly better algorithm.

The use of mutable values, such as vectors and hash tables, raises fewer suspicions about the style of a program than using set! directly. Nevertheless, simply replacing set!s in a program with a vector-set!s obviously does not improve the style of the program.

4.9.2 Multiple Values: set!-values

The set!-values form assigns to multiple variables at once, given an expression that produces an appropriate number of values:

```
(set!-values (id ...) expr)
```

This form is equivalent to using let-values to receive multiple results from expr, and then assigning the results individually to the ids using set!.

Examples:
```
(define game
  (let ([w 0]
        [l 0])
    (lambda (win?)
      (if win?
          (set! w (+ w 1))
          (set! l (+ l 1)))
      (begin0
        (values w l)
        ; swap sides...
        (set!-values (w l) (values l w))))))
> (game #t)
1
0
> (game #t)
1
1
> (game #f)
1
2
```

4.10 Quoting: quote and '

The quote form produces a constant:

```
(quote datum)
```

The syntax of a `datum` is technically specified as anything that the `read` function parses as a single element. The value of the `quote` form is the same value that `read` would produce given `datum`.

The `datum` can be a symbol, a boolean, a number, a (character or byte) string, a character, a keyword, an empty list, a pair (or list) containing more such values, a vector containing more such values, a hash table containing more such values, or a box containing another such value.

Examples:
```
> (quote apple)
'apple
> (quote #t)
#t
> (quote 42)
42
> (quote "hello")
"hello"
> (quote ())
'()
> (quote ((1 2 3) #("z" x) . the-end))
'((1 2 3) #("z" x) . the-end)
> (quote (1 2 . (3)))
'(1 2 3)
```

As the last example above shows, the `datum` does not have to match the normalized printed form of a value. A `datum` cannot be a printed representation that starts with `#<`, so it cannot be `#<void>`, `#<undefined>`, or a procedure.

The `quote` form is rarely used for a `datum` that is a boolean, number, or string by itself, since the printed forms of those values can already be used as constants. The `quote` form is more typically used for symbols and lists, which have other meanings (identifiers, function calls, etc.) when not quoted.

An expression

```
'datum
```

is a shorthand for

```
(quote datum)
```

and this shorthand is almost always used instead of quote. The shorthand applies even within the *datum*, so it can produce a list containing quote.

Examples:
```
> 'apple
'apple
> '"hello"
"hello"
> '(1 2 3)
'(1 2 3)
> (display '(you can 'me))
(you can (quote me))
```

4.11 Quasiquoting: quasiquote and '

The quasiquote form is similar to quote:

```
(quasiquote datum)
```

However, for each (unquote *expr*) that appears within the *datum*, the *expr* is evaluated to produce a value that takes the place of the unquote sub-form.

Example:
```
> (quasiquote (1 2 (unquote (+ 1 2)) (unquote (- 5 1))))
'(1 2 3 4)
```

This form can be used to write functions that build lists according to certain patterns.

Examples:
```
> (define (deep n)
    (cond
      [(zero? n) 0]
      [else
       (quasiquote ((unquote n) (unquote (deep (- n 1)))))]))
> (deep 8)
'(8 (7 (6 (5 (4 (3 (2 (1 0)))))))))
```

Or even to cheaply construct expressions programmatically. (Of course, 9 times out of 10, you should be using a macro to do this (the 10th time being when you're working through a textbook like PLAI).)

Examples:
```
> (define (build-exp n)
    (add-lets n (make-sum n)))
```

89

```
> (define (add-lets n body)
    (cond
      [(zero? n) body]
      [else
       (quasiquote
        (let ([(unquote (n->var n)) (unquote n)])
          (unquote (add-lets (- n 1) body))))]))
> (define (make-sum n)
    (cond
      [(= n 1) (n->var 1)]
      [else
       (quasiquote (+ (unquote (n->var n))
                      (unquote (make-sum (- n 1)))))]))
> (define (n->var n) (string->symbol (format "x~a" n)))
> (build-exp 3)
'(let ((x3 3)) (let ((x2 2)) (let ((x1 1)) (+ x3 (+ x2 x1)))))
```

The unquote-splicing form is similar to unquote, but its *expr* must produce a list, and the unquote-splicing form must appear in a context that produces either a list or a vector. As the name suggests, the resulting list is spliced into the context of its use.

Example:
```
> (quasiquote (1 2 (unquote-splicing (list (+ 1 2) (- 5 1))) 5))
'(1 2 3 4 5)
```

Using splicing we can revise the construction of our example expressions above to have just a single let expression and a single + expression.

Examples:
```
> (define (build-exp n)
    (add-lets
     n
     (quasiquote (+ (unquote-splicing
                     (build-list
                      n
                      (λ (x) (n->var (+ x 1)))))))))
> (define (add-lets n body)
    (quasiquote
     (let (unquote
           (build-list
            n
            (λ (n)
              (quasiquote
               [(unquote (n->var (+ n 1))) (unquote (+ n 1))]))))
       (unquote body))))
> (define (n->var n) (string->symbol (format "x~a" n)))
```

90

```
> (build-exp 3)
'(let ((x1 1) (x2 2) (x3 3)) (+ x1 x2 x3))
```

If a quasiquote form appears within an enclosing quasiquote form, then the inner quasiquote effectively cancels one layer of unquote and unquote-splicing forms, so that a second unquote or unquote-splicing is needed.

Example:
```
> (quasiquote (1 2 (quasiquote (unquote (+ 1 2)
                                (unquote (unquote (- 5 1)))))))
'(1 2 (quasiquote (unquote (+ 1 2)) (unquote 4)))
```

The evaluation above will not actually print as shown. Instead, the shorthand form of quasiquote and unquote will be used: ` (i.e., a backquote) and , (i.e., a comma). The same shorthands can be used in expressions:

Example:
```
> `(1 2 `(,(+ 1 2) ,,(- 5 1)))
'(1 2 `(,(+ 1 2) ,4))
```

The shorthand for of unquote-splicing is ,@:

Example:
```
> `(1 2 ,@(list (+ 1 2) (- 5 1)))
'(1 2 3 4)
```

4.12 Simple Dispatch: `case`

The `case` form dispatches to a clause by matching the result of an expression to the values for the clause:

```
(case expr
  [(datum ...+) expr ...+]
  ...)
```

Each *datum* will be compared to the result of the first *expr* using `eqv?`. Since `eqv?` doesn't work on many kinds of values, notably strings and lists, each *datum* is typically a number, symbol, or boolean.

Multiple *datum*s can be supplied for each clause, and the corresponding *expr* is evaluated of any of the *datum*s match.

Example:
```
> (let ([v (random 6)])
```

```
      (printf "~a\n" v)
      (case v
        [(0) 'zero]
        [(1) 'one]
        [(2) 'two]
        [(3 4 5) 'many]))
  2
  'two
```

The last clause of a `case` form can use `else`, just like `cond`:

Example:
```
  > (case (random 6)
      [(0) 'zero]
      [(1) 'one]
      [(2) 'two]
      [else 'many])
  'one
```

For more general pattern matching, use `match`, which is introduced in §12 "Pattern Matching".

4.13 Dynamic Binding: `parameterize`

The `parameterize` form associates a new value with a *parameter* during the evaluation of *body* expressions:

```
(parameterize ([parameter-expr value-expr] ...)
  body ...+)
```

The term "parameter" is sometimes used to refer to the arguments of a function, but "parameter" in Racket has the more specific meaning described here.

For example, the `error-print-width` parameter controls how many characters of a value are printed in an error message:

```
  > (parameterize ([error-print-width 5])
      (car (expt 10 1024)))
  car: expects argument of type <pair>; given 10...
  > (parameterize ([error-print-width 10])
      (car (expt 10 1024)))
  car: expects argument of type <pair>; given 1000000...
```

More generally, parameters implement a kind of dynamic binding. The `make-parameter` function takes any value and returns a new parameter that is initialized to the given value. Applying the parameter as a function returns its current value:

```
> (define location (make-parameter "here"))
> (location)
"here"
```

In a `parameterize` form, each *parameter-expr* must produce a parameter. During the evaluation of the `body`s, each specified parameter is given the result of the corresponding *value-expr*. When control leaves the `parameterize` form—either through a normal return, an exception, or some other escape—the parameter reverts to its earlier value:

```
> (parameterize ([location "there"])
    (location))
"there"
> (location)
"here"
> (parameterize ([location "in a house"])
    (list (location)
          (parameterize ([location "with a mouse"])
            (location))
          (location)))
'("in a house" "with a mouse" "in a house")
> (parameterize ([location "in a box"])
    (car (location)))
car: expects argument of type <pair>; given "in a box"
> (location)
"here"
```

The `parameterize` form is not a binding form like `let`; each use of `location` above refers directly to the original definition. A `parameterize` form adjusts the value of a parameter during the whole time that the `parameterize` body is evaluated, even for uses of the parameter that are textually outside of the `parameterize` body:

```
> (define (would-you-could-you?)
    (and (not (equal? (location) "here"))
         (not (equal? (location) "there"))))
> (would-you-could-you?)
#f
> (parameterize ([location "on a bus"])
    (would-you-could-you?))
#t
```

If a use of a parameter is textually inside the body of a `parameterize` but not evaluated before the `parameterize` form produces a value, then the use does not see the value installed by the `parameterize` form:

```
> (let ([get (parameterize ([location "with a fox"])
                (lambda () (location)))])
    (get))
```

```
"here"
```

The current binding of a parameter can be adjusted imperatively by calling the parameter as a function with a value. If a `parameterize` has adjusted the value of the parameter, then directly applying the parameter procedure affects only the value associated with the active `parameterize`:

```
> (define (try-again! where)
    (location where))
> (location)
"here"
> (parameterize ([location "on a train"])
    (list (location)
          (begin (try-again! "in a boat")
                 (location))))
'("on a train" "in a boat")
> (location)
"here"
```

Using `parameterize` is generally preferable to updating a parameter value imperatively— for much the same reasons that binding a fresh variable with `let` is preferable to using `set!` (see §4.9 "Assignment: `set!`").

It may seem that variables and `set!` can solve many of the same problems that parameters solve. For example, `lokation` could be defined as a string, and `set!` could be used to adjust its value:

```
> (define lokation "here")
> (define (would-ya-could-ya?)
    (and (not (equal? lokation "here"))
         (not (equal? lokation "there"))))
> (set! location "on a bus")
> (would-ya-could-ya?)
#f
```

Parameters, however, offer several crucial advantages over `set!`:

- The `parameterize` form helps automatically reset the value of a parameter when control escapes due to an exception. Adding exception handlers and other forms to rewind a `set!` is relatively tedious.

- Parameters work nicely with tail calls (see §2.3.3 "Tail Recursion"). The last *body* in a `parameterize` form is in tail position with respect to the `parameterize` form.

- Parameters work properly with threads (see §10.1 "Threads"). The `parameterize` form adjusts the value of a parameter only for evaluation in the current thread, which avoids race conditions with other threads.

94

5   Programmer-Defined Datatypes

New datatypes are normally created with the `struct` form, which is the topic of this chapter. The class-based object system, which we defer to §13 "Classes and Objects", offers an alternate mechanism for creating new datatypes, but even classes and objects are implemented in terms of structure types.

5.1   Simple Structure Types: `struct`

To a first approximation, the syntax of `struct` is

```
(struct struct-id (field-id ...))
```

Examples:
```
(struct posn (x y))
```

The `struct` form binds *struct-id* and a number of identifiers that are built from *struct-id* and the *field-id*s:

- *struct-id* : a *constructor* function that takes as many arguments as the number of *field-id*s, and returns an instance of the structure type.

  Example:
  ```
  > (posn 1 2)
  #<posn>
  ```

- *struct-id?* : a *predicate* function that takes a single argument and returns `#t` if it is an instance of the structure type, `#f` otherwise.

  Examples:
  ```
  > (posn? 3)
  #f
  > (posn? (posn 1 2))
  #t
  ```

- *struct-id-field-id* : for each *field-id*, an *accessor* that extracts the value of the corresponding field from an instance of the structure type.

  Examples:
  ```
  > (posn-x (posn 1 2))
  1
  > (posn-y (posn 1 2))
  2
  ```

- `struct:`*`struct-id`* : a *structure type descriptor*, which is a value that represents the structure type as a first-class value (with `#:super`, as discussed later in §5.8 "More Structure Type Options").

A `struct` form places no constraints on the kinds of values that can appear for fields in an instance of the structure type. For example, `(posn "apple" #f)` produces an instance of `posn`, even though `"apple"` and `#f` are not valid coordinates for the obvious uses of `posn` instances. Enforcing constraints on field values, such as requiring them to be numbers, is normally the job of a contract, as discussed later in §7 "Contracts".

5.2   Copying and Update

The `struct-copy` form clones a structure and optionally updates specified fields in the clone. This process is sometimes called a *functional update*, because the result is a structure with updated field values. but the original structure is not modified.

```
(struct-copy struct-id struct-expr [field-id expr] ...)
```

The *`struct-id`* that appears after `struct-copy` must be a structure type name bound by `struct` (i.e., the name that cannot be used directly as an expression). The *`struct-expr`* must produce an instance of the structure type. The result is a new instance of the structure type that is like the old one, except that the field indicated by each *`field-id`* gets the value of the corresponding *`expr`*.

Examples:
```
> (define p1 (posn 1 2))
> (define p2 (struct-copy posn p1 [x 3]))
> (list (posn-x p2) (posn-y p2))
'(3 2)
> (list (posn-x p1) (posn-x p2))
'(1 3)
```

5.3   Structure Subtypes

An extended form of `struct` can be used to define a *structure subtype*, which is a structure type that extends an existing structure type:

```
(struct struct-id super-id (field-id ...))
```

The *super-id* must be a structure type name bound by `struct` (i.e., the name that cannot be used directly as an expression).

Examples:
```
(struct posn (x y))
(struct 3d-posn posn (z))
```

A structure subtype inherits the fields of its supertype, and the subtype constructor accepts the values for the subtype fields after values for the supertype fields. An instance of a structure subtype can be used with the predicate and accessors of the supertype.

Examples:
```
> (define p (3d-posn 1 2 3))
> p
#<3d-posn>
> (posn? p)
#t
> (posn-x p)
1
> (3d-posn-z p)
3
```

5.4 Opaque versus Transparent Structure Types

With a structure type definition like

```
(struct posn (x y))
```

an instance of the structure type prints in a way that does not show any information about the fields values. That is, structure types by default are *opaque*. If the accessors and mutators of a structure type are kept private to a module, then no other module can rely on the representation of the type's instances.

To make a structure type *transparent*, use the `#:transparent` keyword after the field-name sequence:

```
(struct posn (x y)
        #:transparent)

> (posn 1 2)
(posn 1 2)
```

An instance of a transparent structure type prints like a call to the constructor, so that it shows the structures field values. A transparent structure type also allows reflective operations, such as `struct?` and `struct-info`, to be used on its instances (see §15 "Reflection and Dynamic

Evaluation").

Structure types are opaque by default, because opaque structure instances provide more encapsulation guarantees. That is, a library can use an opaque structure to encapsulate data, and clients of the library cannot manipulate the data in the structure except as allowed by the library.

5.5   Structure Comparisons

A generic `equal?` comparison automatically recurs on the fields of a transparent structure type, but `equal?` defaults to mere instance identity for opaque structure types:

```
(struct glass (width height) #:transparent)

> (equal? (glass 1 2) (glass 1 2))
#t

(struct lead (width height))

> (define slab (lead 1 2))
> (equal? slab slab)
#t
> (equal? slab (lead 1 2))
#f
```

To support instances comparisons via `equal?` without making the structure type transparent, you can use the `#:property` keyword, `prop:equal+hash`, and then a list of three functions:

```
(struct lead (width height)
  #:property
  prop:equal+hash
  (list (lambda (a b equal?-recur)
          ; compare a and b
          (and (equal?-recur (lead-width a) (lead-width b))
               (equal?-recur (lead-height a) (lead-height b))))
        (lambda (a hash-recur)
          ; compute primary hash code of a
          (+ (hash-recur (lead-width a))
             (* 3 (hash-recur (lead-height a)))))
        (lambda (a hash2-recur)
          ; compute secondary hash code of a
          (+ (hash2-recur (lead-width a))
             (hash2-recur (lead-height a))))))
```

```
> (equal? (lead 1 2) (lead 1 2))
#t
```

The first function in the list implements the `equal?` test on two `lead`s; the third argument to the function is used instead of `equal?` for recursive equality testing, so that data cycles can be handled correctly. The other two functions compute primary and secondary hash codes for use with hash tables:

```
> (define h (make-hash))
> (hash-set! h (lead 1 2) 3)
> (hash-ref h (lead 1 2))
3
> (hash-ref h (lead 2 1))
hash-ref: no value found for key: #<lead>
```

The first function provided with `prop:equal+hash` is not required to recursively compare the fields of the structure. For example, a structure type representing a set might implement equality by checking that the members of the set are the same, independent of the order of elements in the internal representation. Just take care that the hash functions produce the same value for any two structure types that are supposed to be equivalent.

5.6 Structure Type Generativity

Each time that a `struct` form is evaluated, it generates a structure type that is distinct from all existing structure types, even if some other structure type has the same name and fields.

This generativity is useful for enforcing abstractions and implementing programs such as interpreters, but beware of placing a `struct` form in positions that are evaluated multiple times.

Examples:
```
(define (add-bigger-fish lst)
  (struct fish (size) #:transparent) ; new every time
  (cond
   [(null? lst) (list (fish 1))]
   [else (cons (fish (* 2 (fish-size (car lst))))
               lst)]))
> (add-bigger-fish null)
(list  (fish 1))
> (add-bigger-fish (add-bigger-fish null))
fish-size: expects args of type <struct:fish>; given
instance of a different <struct:fish>

(struct fish (size) #:transparent)
(define (add-bigger-fish lst)
```

```
  (cond
   [(null? lst) (list (fish 1))]
   [else (cons (fish (* 2 (fish-size (car lst))))
               lst)]))

> (add-bigger-fish (add-bigger-fish null))
(list  (fish 2) (fish 1))
```

5.7   Prefab Structure Types

Although a transparent structure type prints in a way that shows its content, the printed form of the structure cannot be used in an expression to get the structure back, unlike the printed form of a number, string, symbol, or list.

A *prefab* ("previously fabricated") structure type is a built-in type that is known to the Racket printer and expression reader. Infinitely many such types exist, and they are indexed by name, field count, supertype, and other such details. The printed form of a prefab structure is similar to a vector, but it starts `#s` instead of just `#`, and the first element in the printed form is the prefab structure type's name.

The following examples show instances of the `sprout` prefab structure type that has one field. The first instance has a field value `'bean`, and the second has field value `'alfalfa`:

```
> '#s(sprout bean)
'#s(sprout bean)
> '#s(sprout alfalfa)
'#s(sprout alfalfa)
```

Like numbers and strings, prefab structures are "self-quoting," so the quotes above are optional:

```
> #s(sprout bean)
'#s(sprout bean)
```

When you use the `#:prefab` keyword with `struct`, instead of generating a new structure type, you obtain bindings that work with the existing prefab structure type:

```
> (define lunch '#s(sprout bean))
> (struct sprout (kind) #:prefab)
> (sprout? lunch)
#t
> (sprout-kind lunch)
'bean
> (sprout 'garlic)
'#s(sprout garlic)
```

The field name `kind` above does not matter for finding the prefab structure type; only the name `sprout` and the number of fields matters. At the same time, the prefab structure type `sprout` with three fields is a different structure type than the one with a single field:

```
> (sprout? #s(sprout bean #f 17))
#f
> (struct sprout (kind yummy? count) #:prefab) ; redefine
> (sprout? #s(sprout bean #f 17))
#t
> (sprout? lunch)
#f
```

A prefab structure type can have another prefab structure type as its supertype, it can have mutable fields, and it can have auto fields. Variations in any of these dimensions correspond to different prefab structure types, and the printed form of the structure type's name encodes all of the relevant details.

```
> (struct building (rooms [location #:mutable]) #:prefab)
> (struct house building ([occupied #:auto]) #:prefab
    #:auto-value 'no)
> (house 5 'factory)
'#s((house (1 no) building 2 #(1))  5 factory no)
```

Every prefab structure type is transparent—but even less abstract than a transparent type, because instances can be created without any access to a particular structure-type declaration or existing examples. Overall, the different options for structure types offer a spectrum of possibilities from more abstract to more convenient:

- Opaque (the default) : Instances cannot be inspected or forged without access to the structure-type declaration. As discussed in the next section, constructor guards and properties can be attached to the structure type to further protect or to specialize the behavior of its instances.

- Transparent : Anyone can inspect or create an instance without access to the structure-type declaration, which means that the value printer can show the content of an instance. All instance creation passes through a constructor guard, however, so that the content of an instance can be controlled, and the behavior of instances can be specialized through properties. Since the structure type is generated by its definition, instances cannot be manufactured simply through the name of the structure type, and therefore cannot be generated automatically by the expression reader.

- Prefab : Anyone can inspect or create an instance at any time, without prior access to a structure-type declaration or an example instance. Consequently, the expression reader can manufacture instances directly. The instance cannot have a constructor guard or properties.

Since the expression reader can generate prefab instances, they are useful when convenient

serialization is more important than abstraction. Opaque and transparent structures also can be serialized, however, if they are defined with `define-serializable-struct` as described in §8.4 "Datatypes and Serialization".

5.8   More Structure Type Options

The full syntax of `struct` supports many options, both at the structure-type level and at the level of individual fields:

```
(struct struct-id maybe-super (field ...)
        struct-option ...)

maybe-super =
            | super-id

      field = field-id
            | [field-id field-option ...]
```

A *struct-option* always starts with a keyword:

```
#:mutable
```

> Causes all fields of the structure to be mutable, and introduces for each *field-id* a *mutator* `set-struct-id-field-id!` that sets the value of the corresponding field in an instance of the structure type.
> Examples:
> ```
> > (struct dot (x y) #:mutable)
> (define d (dot 1 2))
> > (dot-x d)
> 1
> > (set-dot-x! d 10)
> > (dot-x d)
> 10
> ```
> The `#:mutable` option can also be used as a *field-option*, in which case it makes an individual field mutable.
> Examples:
> ```
> > (struct person (name [age #:mutable]))
> (define friend (person "Barney" 5))
> > (set-person-age! friend 6)
> > (set-person-name! friend "Mary")
> reference to undefined identifier: set-person-name!
> ```

`#:transparent`

> Controls reflective access to structure instances, as discussed in a previous section, §5.4 "Opaque versus Transparent Structure Types".

`#:inspector` *inspector-expr*

> Generalizes `#:transparent` to support more controlled access to reflective operations.

`#:prefab`

> Accesses a built-in structure type, as discussed in a previous section, §5.7 "Prefab Structure Types".

`#:auto-value` *auto-expr*

> Specifies a value to be used for all automatic fields in the structure type, where an automatic field is indicated by the `#:auto` field option. The constructor procedure does not accept arguments for automatic fields. Automatic fields are implicitly mutable (via reflective operations), but mutator functions are bound only if `#:mutator` is also specified.
>
> Examples:
> ```
> > (struct posn (x y [z #:auto])
>                #:transparent
>                #:auto-value 0)
> > (posn 1 2)
> (posn 1 2 0)
> ```

`#:guard` *guard-expr*

> Specifies a *constructor guard* procedure to be called whenever an instance of the structure type is created. The guard takes as many arguments as non-automatic fields in the structure type, plus one more for the name of the instantiated type (in case a sub-type is instantiated, in which case it's best to report an error using the sub-type's name). The guard should return the same number of values as given, minus the name argument. The guard can raise an exception if one of the given arguments is unacceptable, or it can convert an argument.
>
> Examples:
> ```
> > (struct thing (name)
>            #:transparent
> ```

```
               #:guard (lambda (name type-name)
                         (cond
                           [(string? name) name]
                           [(symbol? name) (symbol-
>string name)]
                           [else (error type-name
                                        "bad name: ~e"
                                        name)]))))
  > (thing "apple")
  (thing "apple")
  > (thing 'apple)
  (thing "apple")
  > (thing 1/2)
  thing: bad name: 1/2
```

The guard is called even when subtype instances are created. In that case, only the fields accepted by the constructor are provided to the guard (but the subtype's guard gets both the original fields and fields added by the subtype).

Examples:

```
  > (struct person thing (age)
            #:transparent
            #:guard (lambda (name age type-name)
                      (if (negative? age)
                          (error type-name "bad
age: ~e" age)
                          (values name age))))
  > (person "John" 10)
  (person "John" 10)
  > (person "Mary" -1)
  person: bad age: -1
  > (person 10 10)
  person: bad name: 10
```

`#:property prop-expr val-expr`

Associates a *property* and value with the structure type. For example, the `prop:procedure` property allows a structure instance to be used as a function; the property value determines how a call is implemented when using the structure as a function.

Examples:

```
  > (struct greeter (name)
            #:property prop:procedure
                       (lambda (self other)
                         (string-append
                          "Hi " other
```

```
                                    ", I'm " (greeter-
name self))))
  (define joe-greet (greeter "Joe"))
  > (greeter-name joe-greet)
  "Joe"
  > (joe-greet "Mary")
  "Hi Mary, I'm Joe"
  > (joe-greet "John")
  "Hi John, I'm Joe"
```

`#:super` *super-expr*

An alternative to supplying a `super-id` next to `struct-id`. Instead of the name of a structure type (which is not an expression), *super-expr* should produce a structure type descriptor value. An advantage of `#:super` is that structure type descriptors are values, so they can be passed to procedures.

Examples:

```
  (define (raven-constructor super-type)
    (struct raven ()
            #:super super-type
            #:transparent
            #:property prop:procedure (lambda (self)
                                        'nevermore))
    raven)
  > (let ([r ((raven-constructor struct:posn) 1 2)])
      (list r (r)))
  (list  (raven 1 2) 'nevermore)
  > (let ([r ((raven-constructor struct:thing) "apple")])
      (list r (r)))
  (list  (raven "apple") 'nevermore)
```

§4 "Structures" in **Reference**: *Racket* provides more on structure types.

105

6 Modules

Modules let you organize Racket code into multiple files and reusable libraries.

6.1 Module Basics

The space of module names is distinct from the space of normal Racket definitions. Indeed, since modules typically reside in files, the space of module names is explicitly tied to the filesystem at run time. For example, if the file `"/home/molly/cake.rkt"` contains

```racket
#lang racket

(provide print-cake)

; draws a cake with n candles
(define (print-cake n)
  (printf "   ~a  \n" (make-string n #\.))
  (printf " .-~a-.\n" (make-string n #\|))
  (printf " | ~a |\n" (make-string n #\space))
  (printf "---~a---\n" (make-string n #\-)))
```

then it can be used as the source of a module whose full name is based on the path `"/home/molly/cake.rkt"`. The `provide` line exports the definition `print-cake` so that it can be used outside the module.

Instead of using its full path, a module is more likely to be referenced by a relative path. For example, a file `"/home/molly/random-cake.rkt"` could use the `"cake.rkt"` module like this:

```racket
#lang racket

(require "cake.rkt")

(print-cake (random 30))
```

The relative reference `"cake.rkt"` in the import `(require "cake.rkt")` works because the `"cake.rkt"` module source is in the same directory as the `"random-cake.rkt"` file. (Unix-style relative paths are used for relative module references on all platforms, much like relative URLs.)

Library modules that are distributed with Racket are usually referenced through an unquoted, suffixless path. The path is relative to the library installation directory, which contains directories for individual library *collections*. The module below refers to the `"date.rkt"` library that is part of the `"racket"` collection.

```
#lang racket

(require racket/date)

(printf "Today is ~s\n"
        (date->string (seconds->date (current-seconds))))
```

In addition to the main collection directory, which contains all collections that are part of the installation, collections can also be installed in a user-specific location. Finally, additional collection directories can be specified in configuration files or through the PLTCOLLECTS search path. Try running the following program to find out where your collections are:

```
#lang racket

(require setup/dirs)

(find-collects-dir) ; main collection directory
(find-user-collects-dir) ; user-specific collection directory
(get-collects-search-dirs) ; complete search path
```

We discuss more forms of module reference later in §6.3 "Module Paths".

6.2   Module Syntax

The #lang at the start of a module file begins a shorthand for a module form, much like ' is a shorthand for a quote form. Unlike ', the #lang shorthand does not work well in a REPL, in part because it must be terminated by an end-of-file, but also because the longhand expansion of #lang depends on the name of the enclosing file.

6.2.1   The module Form

The longhand form of a module declaration, which works in a REPL as well as a file, is

```
(module name-id initial-module-path
  decl ...)
```

where the *name-id* is a name for the module, *initial-module-path* is an initial import, and each *decl* is an import, export, definition, or expression. In the case of a file, *name-id* must match the name of the containing file, minus its directory path or file extension.

The *initial-module-path* is needed because even the require form must be imported for further use in the module body. In other words, the *initial-module-path* import boot-

straps the syntax available in the body. The most commonly used *initial-module-path* is racket, which supplies most of the bindings described in this guide, including require, define, and provide. Another commonly used *initial-module-path* is racket/base, which provides less functionality, but still much of the most commonly needed functions and syntax.

For example, the "cake.rkt" example of the previous section could be written as

```
(module cake racket
  (provide print-cake)

  (define (print-cake n)
    (printf "   ~a  \n" (make-string n #\.))
    (printf " .-~a-.\n" (make-string n #\|))
    (printf " | ~a |\n" (make-string n #\space))
    (printf "---~a---\n" (make-string n #\-))))
```

Furthermore, this module form can be evaluated in a REPL to declare a cake module that is not associated with any file. To refer to such an unassociated module, quote the module name:

Examples:
```
> (require 'cake)
> (print-cake 3)
     ...
  .-|||-.
  |     |
 ---------
```

Declaring a module does not immediately evaluate the body definitions and expressions of the module. The module must be explicitly required at the top level to trigger evaluation. After evaluation is triggered once, later requires do not re-evaluate the module body.

Examples:
```
> (module hi racket
    (printf "Hello\n"))
> (require 'hi)
Hello
> (require 'hi)
```

6.2.2 The #lang Shorthand

The body of a #lang shorthand has no specific syntax, because the syntax is determined by the language name that follows #lang.

In the case of #lang racket, the syntax is

```
#lang racket
decl ...
```

which reads the same as

```
(module name racket
  decl ...)
```

where *name* is derived from the name of the file that contains the #lang form.

The #lang racket/base form has the same syntax as #lang racket, except that the long-hand expansion uses racket/base instead of racket. The #lang honu form, in contrast, has a completely different syntax that doesn't even look like Racket, and which we do not attempt to describe in this guide.

Unless otherwise specified, a module that is documented as a "language" using the #lang notation will expand to module in the same way as #lang racket. The documented language name can be used directly with module or require, too.

6.3 Module Paths

A *module path* is a reference to a module, as used with require or as the *initial-module-path* in a module form. It can be any of several forms:

`(quote id)`

> A module path that is a quoted identifier refers to a non-file module declaration using the identifier. This form of module reference makes the most sense in a REPL.
>
> Examples:
> ```
> > (module m racket
>     (provide color)
>     (define color "blue"))
> > (module n racket
>     (require 'm)
>     (printf "my favorite color is ~a\n" color))
> > (require 'n)
> my favorite color is blue
> ```

`rel-string`

A string module path is a relative path using Unix-style conventions: `/` is the path separator, `..` refers to the parent directory, and `.` refers to the same directory. The *rel-string* must not start or end with a path separator. If the path has no suffix, `".rkt"` is added automatically.

The path is relative to the enclosing file, if any, or it is relative to the current directory. (More precisely, the path is relative to the value of (`current-load-relative-directory`), which is set while loading a file.)

§6.1 "Module Basics" shows examples using relative paths.

If a relative path ends with a `".ss"` suffix, it is converted to `".rkt"`. If the file that implements the referenced module actually ends in `".ss"`, the suffix will be changed back when attempting to load the file (but a `".rkt"` suffix takes precedence). This two-way conversion provides compatibility with older versions of Racket.

---

id

A module path that is an unquoted identifier refers to an installed library. The *id* is constrained to contain only ASCII letters, ASCII numbers, `+`, `-`, `_`, and `/`, where `/` separates path elements within the identifier. The elements refer to collections and sub-collections, instead of directories and sub-directories.

An example of this form is `racket/date`. It refers to the module whose source is the `"date.rkt"` file in the `"racket"` collection, which is installed as part of Racket. The `".rkt"` suffix is added automatically.

Another example of this form is `racket`, which is commonly used at the initial import. The path `racket` is shorthand for `racket/main`; when an *id* has no `/`, then `/main` is automatically added to the end. Thus, `racket` or `racket/main` refers to the module whose source is the `"main.rkt"` file in the `"racket"` collection.

Examples:
```
> (module m racket
    (require racket/date)

    (printf "Today is ~s\n"
            (date->string (seconds->date (current-
seconds)))))
> (require 'm)
Today is "Monday, February 14th, 2011"
```

When the full path of a module ends with `".rkt"`, if no such file exists but one does exist with the `".ss"` suffix, then the `".ss"` suffix is substituted automatically. This transformation provides compatibility with older versions of Racket.

```
(lib rel-string)
```

Like an unquoted-identifier path, but expressed as a string instead of an identifier. Also, the `rel-string` can end with a file suffix, in which case `".rkt"` is not automatically added.

Example of this form include `(lib "racket/date.rkt")` and `(lib "racket/date")`, which are equivalent to `racket/date`. Other examples include `(lib "racket")`, `(lib "racket/main")`, and `(lib "racket/main.rkt")`, which are all equivalent to `racket`.

Examples:
```
> (module m (lib "racket")
    (require (lib "racket/date.rkt"))

    (printf "Today is ~s\n"
            (date->string (seconds->date (current-
seconds)))))
> (require 'm)
Today is "Monday, February 14th, 2011"
```

```
(planet id)
```

Accesses a third-party library that is distributed through the PLaneT server. The library is downloaded the first time that it is needed, and then the local copy is used afterward.

The `id` encodes several pieces of information separated by a `/`: the package owner, then package name with optional version information, and an optional path to a specific library with the package. Like `id` as shorthand for a `lib` path, a `".rkt"` suffix is added automatically, and `/main` is used as the path if no sub-path element is supplied.

Examples:
```
> (module m (lib "racket")
    ; Use "schematics"'s "random.plt" 1.0, file
"random.rkt":
    (require (planet schematics/random:1/random))
    (display (random-gaussian)))
> (require 'm)
0.9050686838895684
```

As with other forms, an implementation file ending with `".ss"` can be substituted automatically if no implementation file ending with `".rkt"` exists.

```
(planet package-string)
```

Like the symbol form of a `planet`, but using a string instead of an identifier. Also, the *package-string* can end with a file suffix, in which case `".rkt"` is not added.

As with other forms, an `".ss"` extension is converted to `".rkt"`, while an implementation file ending with `".ss"` can be substituted automatically if no implementation file ending with `".rkt"` exists.

```
(planet rel-string (user-string pkg-string vers ...))

vers = nat
     | (nat nat)
     | (= nat)
     | (+ nat)
     | (- nat)
```

A more general form to access a library from the PLaneT server. In this general form, a PLaneT reference starts like a `lib` reference with a relative path, but the path is followed by information about the producer, package, and version of the library. The specified package is downloaded and installed on demand.

The *vers*es specify a constraint on the acceptable version of the package, where a version number is a sequence of non-negative integers, and the constraints determine the allowable values for each element in the sequence. If no constraint is provided for a particular element, then any version is allowed; in particular, omitting all *vers*es means that any version is acceptable. Specifying at least one *vers* is strongly recommended.

For a version constraint, a plain *nat* is the same as `(+ nat)`, which matches *nat* or higher for the corresponding element of the version number. A `(start-nat end-nat)` matches any number in the range *start-nat* to *end-nat*, inclusive. A `(= nat)` matches only exactly *nat*. A `(- nat)` matches *nat* or lower.

Examples:
```
> (module m (lib "racket")
    (require (planet "random.rkt" ("schematics" "random.plt" 1 0)))
    (display (random-gaussian)))
> (require 'm)
0.9050686838895684
```

The automatic `".ss"` and `".rkt"` conversions apply as with other forms.

```
(file string)
```

Refers to a file, where *string* is a relative or absolute path using the current platform's conventions. This form is not portable, and it should *not* be used when a plain, portable `rel-string` suffices.

The automatic ".ss" and ".rkt" conversions apply as with other forms.

6.4 Imports: require

The require form imports from another module. A require form can appear within a module, in which case it introduces bindings from the specified module into importing module. A require form can also appear at the top level, in which case it both imports bindings and *instantiates* the specified module; that is, it evaluates the body definitions and expressions of the specified module, if they have not been evaluated already.

A single require can specify multiple imports at once:

```
(require require-spec ...)
```

Specifying multiple require-specs in a single require is essentially the same as using multiple requires, each with a single require-spec. The difference is minor, and confined to the top-level: a single require can import a given identifier at most once, whereas a separate require can replace the bindings of a previous require (both only at the top level, outside of a module).

The allowed shape of a require-spec is defined recursively:

```
module-path
```

In its simplest form, a require-spec is a module-path (as defined in the previous section, §6.3 "Module Paths"). In this case, the bindings introduced by require are determined by provide declarations within each module referenced by each module-path.

Examples:
```
> (module m racket
    (provide color)
    (define color "blue"))
> (module n racket
    (provide size)
    (define size 17))
> (require 'm 'n)
> (list color size)
'("blue" 17)
```

```
(only-in require-spec id-maybe-renamed ...)
```

```
id-maybe-renamed = id
                 | [orig-id bind-id]
```
An `only-in` form limits the set of bindings that would be introduced by a base `require-spec`. Also, `only-in` optionally renames each binding that is preserved: in a `[orig-id bind-id]` form, the `orig-id` refers to a binding implied by `require-spec`, and `bind-id` is the name that will be bound in the importing context instead of `bind-id`.

Examples:
```
> (module m (lib "racket")
    (provide tastes-great?
             less-filling?)
    (define tastes-great? #t)
    (define less-filling? #t))
> (require (only-in 'm tastes-great?))
> tastes-great?
#t
> less-filling?
reference to undefined identifier: less-filling?
> (require (only-in 'm [less-filling? lite?]))
> lite?
#t
```

`(except-in require-spec id ...)`

This form is the complement of `only`: it excludes specific bindings from the set specified by `require-spec`.

`(rename-in require-spec [orig-id bind-id] ...)`

This form supports renaming like `only-in`, but leaving alone identifiers from `require-spec` that are not mentioned as an `orig-id`.

`(prefix-in prefix-id require-spec)`

This is a shorthand for renaming, where `prefix-id` is added to the front of each identifier specified by `require-spec`.

The `only-in`, `except-in`, `rename-in`, and `prefix-in` forms can be nested to implement more complex manipulations of imported bindings. For example,

```
(require (prefix-in m: (except-in 'm ghost)))
```

imports all bindings that `m` exports, except for the `ghost` binding, and with local names that are prefixed with `m:`.

Equivalently, the `prefix-in` could be applied before `except-in`, as long as the omission with `except-in` is specified using the `m:` prefix:

```
(require (except-in (prefix m: 'm) m:ghost))
```

6.5 Exports: `provide`

By default, all of a module's definitions are private to the module. The `provide` form specifies definitions to be made available where the module is `required`.

```
(provide provide-spec ...)
```

A `provide` form can only appear at module level (i.e., in the immediate body of a `module`). Specifying multiple `provide-spec`s in a single `provide` is exactly the same as using multiple `provide`s each with a single `provide-spec`.

Each identifier can be exported at most once from a module across all `provide`s within the module. More precisely, the external name for each export must be distinct; the same internal binding can be exported multiple times with different external names.

The allowed shape of a `provide-spec` is defined recursively:

```
identifier
```

> In its simplest form, a `provide-spec` indicates a binding within its module to be exported. The binding can be from either a local definition, or from an import.

```
(rename-out [orig-id export-id] ...)
```

> A `rename-out` form is similar to just specifying an identifier, but the exported binding `orig-id` is given a different name, `export-id`, to importing modules.

```
(struct-out struct-id)
```

> A `struct-out` form exports the bindings created by `(struct struct-id ....)`.

See §5 "Programmer-Defined Datatypes" for information on `define-struct`.

```
(all-defined-out)
```

115

The `all-defined-out` shorthand exports all bindings that are defined within the exporting module (as opposed to imported).

Use of the `all-defined-out` shorthand is generally discouraged, because it makes less clear the actual exports for a module, and because Racket programmers get into the habit of thinking that definitions can be added freely to a module without affecting its public interface (which is not the case when `all-defined-out` is used).

`(all-from-out module-path)`

The `all-from-out` shorthand exports all bindings in the module that were imported using a `require-spec` that is based on `module-path`.

Although different `module-path`s could refer to the same file-based module, re-exporting with `all-from-out` is based specifically on the `module-path` reference, and not the module that is actually referenced.

`(except-out provide-spec id ...)`

Like `provide-spec`, but omitting the export of each `id`, where `id` is the external name of the binding to omit.

`(prefix-out prefix-id provide-spec)`

Like `provide-spec`, but adding `prefix-id` to the beginning of the external name for each exported binding.

6.6 Assignment and Redefinition

The use of `set!` on variables defined within a module is limited to the body of the defining module. That is, a module is allowed to change the value of its own definitions, and such changes are visible to importing modules. However, an importing context is not allowed to change the value of an imported binding.

Examples:
```
> (module m racket
    (provide counter increment!)
    (define counter 0)
    (define (increment!)
      (set! counter (add1 counter)))))
> (require 'm)
> counter
```

```
0
> (increment!)
> counter
1
> (set! counter -1)
```
set!: cannot mutate module-required identifier in: counter

As the above example illustrates, a module can always grant others the ability to change its exports by providing a mutator function, such as `increment!`.

The prohibition on assignment of imported variables helps support modular reasoning about programs. For example, in the module,

```
(module m racket
  (provide rx:fish fishy-string?)
  (define rx:fish #rx"fish")
  (define (fishy-string? s)
    (regexp-match? s rx:fish)))
```

the function `fishy-string?` will always match strings that contain "fish", no matter how other modules use the `rx:fish` binding. For essentially the same reason that it helps programmers, the prohibition on assignment to imports also allows many programs to be executed more efficiently.

Along the same lines, when a module contains no `set!` of a particular identifier that is defined within the module, then the identifier is considered a *constant* that cannot be changed—not even by re-declaring the module.

Consequently, re-declaration of a module is not generally allowed. For file-based modules, simply changing the file does not lead to a re-declaration in any case, because file-based modules are loaded on demand, and the previously loaded declarations satisfy future requests. It is possible to use Racket's reflection support to re-declare a module, however, and non-file modules can be re-declared in the REPL; in such cases, the re-declaration may fail if it involves the re-definition of a previously constant binding.

```
> (module m racket
    (define pie 3.141597))
> (require 'm)
> (module m racket
    (define pie 3))
```
define-values: cannot re-define a constant: pie in module:
'm

For exploration and debugging purposes, the Racket reflective layer provides a `compile-enforce-module-constants` parameter to disable the enforcement of constants.

```
> (compile-enforce-module-constants #f)
```

117

```
> (module m2 racket
    (provide pie)
    (define pie 3.141597))
> (require 'm2)
> (module m2 racket
    (provide pie)
    (define pie 3))
> (compile-enforce-module-constants #t)
> pie
3
```

7 Contracts

This chapter provides a gentle introduction to Racket's contract system.

7.1 Contracts and Boundaries

Like a contract between two business partners, a software contract is an agreement between two parties. The agreement specifies obligations and guarantees for each "product" (or value) that is handed from one party to the other.

A contract thus establishes a boundary between the two parties. Whenever a value crosses this boundary, the contract monitoring system performs contract checks, making sure the partners abide by the established contract.

In this spirit, Racket encourages contracts mainly at module boundaries. Specifically, programmers may attach contracts to `provide` clauses and thus impose constraints and promises on the use of exported values. For example, the export specification

```racket
#lang racket

(provide/contract
  [amount positive?])
(define amount ...)
```

promises to all clients of the above module that the value of `amount` will always be a positive number. The contract system monitors the module's obligation carefully. Every time a client refers to `amount`, the monitor checks that the value of `amount` is indeed a positive number.

The contracts library is built into the Racket language, but if you wish to use `racket/base`, you can explicitly require the contracts library like this:

```racket
#lang racket/base
(require racket/contract) ; now we can write contracts

(provide/contract
  [amount positive?])
(define amount ...)
```

7.1.1 Contract Violations

If we bind `amount` to a number that is not positive,

```racket
#lang racket
```

119

```
(provide/contract
  [amount positive?])
(define amount 0)
```

then, when the module is required, the monitoring system signals a violation of the contract and blames the module for breaking its promises.

An even bigger mistake would be to bind amount to a non-number value:

```
#lang racket

(provide/contract
  [amount positive?])
(define amount 'amount)
```

In this case, the monitoring system will apply positive? to a symbol, but positive? reports an error, because its domain is only numbers. To make the contract capture our intentions for all Racket values, we can ensure that the value is both a number and is positive, combining the two contracts with and/c:

```
(provide/contract
  [amount (and/c number? positive?)])
```

7.1.2 Experimenting with Contracts and Modules

All of the contracts and module in this chapter (excluding those just following) are written using the standard #lang syntax for describing modules. Since modules serve as the boundary between parties in a contract, examples involve multiple modules.

To experiment with multiple modules within a single module or within DrRacket's definitions area, use the racket/load language. The contents of such a module can be other modules (and require statements), using the longhand parenthesized syntax for a module (see §6.2.1 "The module Form"). For example, try the example earlier in this section as follows:

```
#lang racket/load

(module m racket
  (provide/contract [amount (and/c number? positive?)])
  (define amount 150))

(module n racket
  (require 'm)
  (+ amount 10))
```

```
(require 'n)
```

Each of the modules and their contracts are wrapped in parentheses with the `module` key-word at the front. The first form after `module` is the name of the module to be used in a subsequent `require` statement (where each reference through a `require` prefixes the name with a quote). The second form after `module` is the language, and the remaining forms are the body of the module. After all of the modules, a `require` starts one of the modules plus anything that is `require`s.

7.2   Simple Contracts on Functions

A mathematical function has a *domain* and a *range*. The domain indicates the kind of values that the function can accept as arguments, and the range indicates the kind of values that it produces. The conventional notation for a describing a function with its domain and range is

```
f : A -> B
```

where `A` is the domain of the function and `B` is the range.

Functions in a programming language have domains and ranges, too, and a contract can en-sure that a function receives only values in its range and produces only values in its domain. A `->` creates such a contract for a function. The forms after a `->` specify contracts for the domains and finally a contract for the range.

Here is a module that might represent a bank account:

```
#lang racket

(provide/contract
  [deposit (-> number? any)]
  [balance (-> number?)])

(define amount 0)
(define (deposit a) (set! amount (+ amount a)))
(define (balance) amount)
```

The module exports two functions:

- `deposit`, which accepts a number and returns some value that is not specified in the contract, and

- `balance`, which returns a number indicating the current balance of the account.

When a module exports a function, it establishes two channels of communication between

itself as a "server" and the "client" module that imports the function. If the client module calls the function, it sends a value into the server module. Conversely, if such a function call ends and the function returns a value, the server module sends a value back to the client module. This client–server distinction is important, because when something goes wrong, one or the other of the parties is to blame.

If a client module were to apply `deposit` to `'millions`, it would violate the contract. The contract-monitoring system would catch this violation and blame client for breaking the contract with the above module. In contrast, if the `balance` function were to return `'broke`, the contract-monitoring system would blame the server module.

A `->` by itself is not a contract; it is a *contract combinator*, which combines other contracts to form a contract.

7.2.1   Styles of `->`

If you are used to mathematical function, you may prefer a contract arrow to appear between the domain and the range of a function, not at the beginning. If you have read *How to Design Programs*, you have seen this many times. Indeed, you may have seen contracts such as these in other people's code:

```
(provide/contract
  [deposit (number? . -> . any)])
```

If a Racket S-expression contains two dots with a symbol in the middle, the reader rearranges the S-expression and place the symbol at the front, as described in §2.4.3 "Lists and Racket Syntax". Thus,

```
(number? . -> . any)
```

is just another way of writing

```
(-> number? any)
```

7.2.2   `any` **and** `any/c`

The `any` contract used for `deposit` matches any kind of result, and it can only be used in the range position of a function contract. Instead of `any` above, we could use the more specific contract `void?`, which says that the function will always return the `(void)` value. The `void?` contract, however, would require the contract monitoring system to check the return value every time the function is called, even though the "client" module can't do much with the value. In contrast, `any` tells the monitoring system *not* to check the return value, it tells a potential client that the "server" module *makes no promises at all* about the function's return value, even whether it is a single value or multiple values.

The any/c contract is similar to any, in that it makes no demands on a value. Unlike any, any/c indicates a single value, and it is suitable for use as an argument contract. Using any/c as a range contract imposes a check that the function produces a single value. That is,

```racket
(-> integer? any)
```

describes a function that accepts and integer and returns any number of values, while

```racket
(-> integer? any/c)
```

describes a function that accepts an integer and produces a single result (but does not say anything more about the result). The function

```racket
(define (f x) (values (+ x 1) (- x 1)))
```

matches (-> integer? any), but not (-> integer? any/c).

Use any/c as a result contract when it is particularly important to promise a single result from a function. Use any when you want to promise as little as possible (and incur as little checking as possible) for a function's result.

7.2.3 Rolling Your Own Contracts

The deposit function adds the given number to the value of amount. While the function's contract prevents clients from applying it to non-numbers, the contract still allows them to apply the function to complex numbers, negative numbers, or inexact numbers, none of which sensibly represent amounts of money.

The contract system allows programmers to define their own contracts as functions:

```racket
#lang racket

(define (amount? a)
  (and (number? a) (integer? a) (exact? a) (>= a 0)))

(provide/contract
  ; an amount is a natural number of cents
  ; is the given number an amount?
  [deposit (-> amount? any)]
  [amount? (-> any/c boolean?)]
  [balance (-> amount?)])

(define amount 0)
(define (deposit a) (set! amount (+ amount a)))
(define (balance) amount)
```

This module define an `amount?` function as uses it as a contract within `->` contracts. When a client calls the `deposit` function as exported with the contract `(-> amount? any)`, it must supply an exact, nonnegative integer, otherwise the `amount?` function applied to the argument will return `#f`, which will cause the contract-monitoring system to blame the client. Similarly, the server module must provide an exact, nonnegative integer as the result of `balance` to remain blameless.

Of course, it makes no sense to restrict a channel of communication to values that the client doesn't understand. Therefore the module also exports the `amount?` predicate itself, with a contract saying that it accepts an arbitrary value and returns a boolean.

In this case, we could also have used `natural-number/c` in place of `amount?`, since it implies exactly the same check:

```racket
(provide/contract
  [deposit (-> natural-number/c any)]
  [balance (-> natural-number/c)])
```

Every function that accepts one argument can be treated as a predicate and thus used as a contract. For combining existing checks into a new one, however, contract combinators such as `and/c` and `or/c` are often useful. For example, here is yet another way to write the contracts above:

```racket
(define amount/c
  (and/c number? integer? exact? (or/c positive? zero?)))

(provide/contract
  [deposit (-> amount/c any)]
  [balance (-> amount/c)])
```

Other values also serve double duty as contracts. For example, if a function accepts a number or `#f`, `(or/c number? #f)` suffices. Similarly, the `amount/c` contract could have been written with a `0` in place of `zero?`. If you use a regular expression as a contract, the contract accepts strings and byte strings that match the regular expression.

Naturally, you can mix your own contract-implementing functions with combinators like `and/c`. Here is a module for creating strings from banking records:

```racket
#lang racket

(define (has-decimal? str)
  (define L (string-length str))
  (and (>= L 3)
       (char=? #\. (string-ref str (- L 3)))))

(provide/contract
  ; convert a random number to a string
```

```
[format-number (-> number? string?)]

; convert an amount into a string with a decimal
; point, as in an amount of US currency
[format-nat (-> natural-number/c
                (and/c string? has-decimal?))])
```

The contract of the exported function `format-number` specifies that the function consumes a number and produces a string. The contract of the exported function `format-nat` is more interesting than the one of `format-number`. It consumes only natural numbers. Its range contract promises a string that has a `.` in the third position from the right.

If we want to strengthen the promise of the range contract for `format-nat` so that it admits only strings with digits and a single dot, we could write it like this:

```
#lang racket

(define (digit-char? x)
  (member x '(#\1 #\2 #\3 #\4 #\5 #\6 #\7 #\8 #\9 #\0)))

(define (has-decimal? str)
  (define L (string-length str))
  (and (>= L 3)
       (char=? #\. (string-ref str (- L 3)))))

(define (is-decimal-string? str)
  (define L (string-length str))
  (and (has-decimal? str)
       (andmap digit-char?
               (string->list (substring str 0 (- L 3))))
       (andmap digit-char?
               (string->list (substring str (- L 2) L)))))

....

(provide/contract
  ....
  ; convert an amount (natural number) of cents
  ; into a dollar based string
  [format-nat (-> natural-number/c
                  (and/c string?
                         is-decimal-string?))])
```

Alternately, in this case, we could use a regular expression as a contract:

```
#lang racket
```

```
(provide/contract
  ....
  ; convert an amount (natural number) of cents
  ; into a dollar based string
  [format-nat (-> natural-number/c
                  (and/c string?
                         #rx"[0-9]*\\.[0-9][0-9][0-9]"))])
```

7.2.4 Contracts on Higher-order Functions

Function contracts are not just restricted to having simple predicates on their domains or
ranges. Any of the contract combinators discussed here, including function contracts them-
selves, can be used as contracts on the arguments and results of a function.

For example,

```
(-> integer? (-> integer? integer?))
```

is a contract that describes a curried function. It matches functions that accept one argument
and then return another function accepting a second argument before finally returning an
integer. If a server exports a function `make-adder` with this contract, and if `make-adder`
returns a value other than a function, then the server is to blame. If `make-adder` does return
a function, but the resulting function is applied to a value other than an integer, then the
client is to blame.

Similarly, the contract

```
(-> (-> integer? integer?) integer?)
```

describes functions that accept other functions as its input. If a server exports a function
`twice` with this contract and the `twice` is applied to a value other than a function of one
argument, then the client is to blame. If `twice` is applied to a function of one argument and
`twice` calls the give function on a value other than an integer, then the server is to blame.

7.2.5 Contract Messages with "???"

You wrote your module. You added contracts. You put them into the interface so that client
programmers have all the information from interfaces. It's a piece of art:

```
#lang racket

(provide/contract
 [deposit (-> (lambda (x)
```

```
                  (and (number? x) (integer? x) (>= x 0)))
              any)])

(define this 0)
(define (deposit a) ...)
```

Several clients used your module. Others used their modules in turn. And all of a sudden one of them sees this error message:

> *bank-client broke the contract (-> ??? any) it had with myaccount on deposit; expected <???>, given: -10*

Clearly, `bank-client` is a module that uses `myaccount` but what is the *???* doing there? Wouldn't it be nice if we had a name for this class of data much like we have string, number, and so on?

For this situation, Racket provides *flat named contracts*. The use of "contract" in this term shows that contracts are first-class values. The "flat" means that the collection of data is a subset of the built-in atomic classes of data; they are described by a predicate that consumes all Racket values and produces a boolean. The "named" part says what we want to do, which is to name the contract so that error messages become intelligible:

```
#lang racket

(define (amount? x) (and (number? x) (integer? x) (>= x 0)))
(define amount (flat-named-contract 'amount amount?))

(provide/contract
 [deposit (amount . -> . any)])

(define this 0)
(define (deposit a) ...)
```

With this little change, the error message becomes all of the sudden quite readable:

> *bank-client broke the contract (-> amount any) it had with myaccount on deposit; expected <amount>, given: -10*

7.3   Contracts on Functions in General

The `->` contract constructor works for functions that take a fixed number of arguments and where the result contract is independent of the input arguments. To support other kinds of functions, Racket supplies additional contract constructors, notably `->*` and `->i`.

7.3.1 Optional Arguments

Take a look at this excerpt from a string-processing module, inspired by the Scheme cookbook:

```racket
#lang racket

(provide/contract
 ; pad the given str left and right with
 ; the (optional) char so that it is centered
 [string-pad-center (->* (string? natural-number/c)
                         (char?)
                         string?)])

(define (string-pad-center str width [pad #\space])
  (define field-width (min width (string-length str)))
  (define rmargin (ceiling (/ (- width field-width) 2)))
  (define lmargin (floor (/ (- width field-width) 2)))
  (string-append (build-string lmargin (λ (x) pad))
                 str
                 (build-string rmargin (λ (x) pad))))
```

The module exports `string-pad-center`, a function that creates a string of a given `width` with the given string in the center. The default fill character is `#\space`; if the client module wishes to use a different character, it may call `string-pad-center` with a third argument, a `char`, overwriting the default.

The function definition uses optional arguments, which is appropriate for this kind of functionality. The interesting point here is the formulation of the contract for the `string-pad-center`.

The contract combinator `->*`, demands several groups of contracts:

- The first one is a parenthesized group of contracts for all required arguments. In this example, we see two: `string?` and `natural-number/c`.

- The second one is a parenthesized group of contracts for all optional arguments: `char?`.

- The last one is a single contract: the result of the function.

Note if a default value does not satisfy a contract, you won't get a contract error for this interface. If you can't trust yourself to get the initial value right, you need to communicate the initial value across a boundary.

7.3.2 Rest Arguments

The `max` operator consumes at least one real number, but it accepts any number of additional arguments. You can write other such functions using a "rest" argument, such as in `max-abs`:

```
(define (max-abs n . rst)
  (foldr (lambda (n m) (max (abs n) m)) (abs n) rst))
```

Describing this function through a contract requires a further extension of `->*`: a `#:rest` keyword specifies a contract on a list of arguments after the required and optional arguments:

```
(provide/contract
 [max-abs (->* (real?) () #:rest (listof real?) real?)])
```

As always for `->*`, the contracts for the required arguments are enclosed in the first pair of parentheses, which in this case is a single real number. The empty pair of parenthesis indicates that there are no optional arguments (not counting the rest arguments). The contract for the rest argument follows `#:rest`; since all additional arguments must be real numbers, the list of rest arguments must satisfy the contract `(listof real?)`.

7.3.3 Keyword Arguments

It turns out that the `->` contract constructor also contains support for keyword arguments. For example, consider this function, which creates a simple GUI and asks the user a yes-or-no question:

```
#lang racket/gui

(define (ask-yes-or-no-question question
                                #:default answer
                                #:title title
                                #:width w
                                #:height h)
  (define d (new dialog% [label title] [width w] [height h]))
  (define msg (new message% [label question] [parent d]))
  (define (yes) (set! answer #t) (send d show #f))
  (define (no) (set! answer #f) (send d show #f))
  (define yes-b (new button%
                     [label "Yes"] [parent d]
                     [callback (λ (x y) (yes))]
                     [style (if answer '(border) '())]))
  (define no-b (new button%
                    [label "No"] [parent d]
                    [callback (λ (x y) (no))]
                    [style (if answer '() '(border))]))
```

129

```
    (send d show #t)
    answer)

(provide/contract
 [ask-yes-or-no-question
  (-> string?
      #:default boolean?
      #:title string?
      #:width exact-integer?
      #:height exact-integer?
      boolean?)])
```

The contract for `ask-yes-or-no-question` uses `->`, and in the same way that `lambda` (or `define`-based functions) allows a keyword to precede a functions formal argument, `->` allows a keyword to precede a function contract's argument contract. In this case, the contract says that `ask-yes-or-no-question` must receive four keyword arguments, one for each of the keywords `#:default`, `#:title`, `#:width`, and `#:height`. As in a function definition, the order of the keywords in `->` relative to each other does not matter for clients of the function; only the relative order of argument contracts without keywords matters.

If you really want to ask a yes-or-no question via a GUI, you should use `message-box/custom`. For that matter, it's usually better to provide buttons with more specific answers than "yes" and "no."

7.3.4 Optional Keyword Arguments

Of course, many of the parameters in `ask-yes-or-no-question` (from the previous question) have reasonable defaults and should be made optional:

```
(define (ask-yes-or-no-question question
                                #:default answer
                                #:title [title "Yes or No?"]
                                #:width [w 400]
                                #:height [h 200])
   ...)
```

To specify this function's contract, we need to use `->*` again. It supports keywords just as you might expect in both the optional and mandatory argument sections. In this case, we have the mandatory keyword `#:default` and optional keywords `#:title`, `#:width`, and `#:height`. So, we write the contract like this:

```
(provide/contract
 [ask-yes-or-no-question
  (->* (string?
        #:default boolean?)

       (#:title string?
        #:width exact-integer?
        #:height exact-integer?)
```

```
    boolean?)])
```

That is, we put the mandatory keywords in the first section, and we put the optional ones in the second section.

7.3.5   Contracts for `case-lambda`

A function defined with `case-lambda` might impose different constraints on its arguments depending on how many are provided. For example, a `report-cost` function might convert either a pair or numbers or a string into a new string:

```
(define report-cost
  (case-lambda
    [(lo hi) (format "between $~a and $~a" lo hi)]
    [(desc) (format "~a of dollars" desc)]))

> (report-cost 5 8)
"between $5 and $8"
> (report-cost "millions")
"millions of dollars"
```

The contract for such a function is formed with the `case->` combinator, which combines as many functional contracts as needed:

```
(provide/contract
  [report-cost
   (case->
    (integer? integer? . -> . string?)
    (string? . -> . string?))])
```

As you can see, the contract for `report-cost` combines two function contracts, which is just as many clauses as the explanation of its functionality required.

7.3.6   Argument and Result Dependencies

The following is an excerpt from an imaginary numerics module:

```
(provide/contract
 [real-sqrt (->i ([argument (>=/c 1)])
                 [result (argument) (<=/c argument)])])
```

The contract for the exported function `real-sqrt` uses the `->i` rather than `->*` function contract. The "i" stands for an *indy dependent* contract, meaning the contract for the function

131

range depends on the value of the argument. The appearance of `argument` in the line for `result`'s contract means that the result depends on the argument. In this particular case, the argument of `real-sqrt` is greater or equal to 1, so a very basic correctness check is that the result is smaller than the argument.

In general, a dependent function contract looks just like the more general `->*` contract, but with names added that can be used elsewhere in the contract.

Going back to the back-account example, suppose that we generalize the module to support multiple accounts and that we also include a withdrawal operation. The improved bank-account module includes a `account` structure type and the following functions:

```
(provide/contract
 [balance (-> account? amount/c)]
 [withdraw (-> account? amount/c account?)]
 [deposit (-> account? amount/c account?)])
```

Besides requiring that a client provide a valid amount for a withdrawal, however, the amount should be less than the specified account's balance, and the resulting account will have less money than it started with. Similarly, the module might promise that a deposit produces an account with money added to the account. The following implementation enforces those constraints and guarantees through contracts:

```
#lang racket

; section 1: the contract definitions
(struct account (balance))
(define amount/c natural-number/c)

; section 2: the exports
(provide/contract
 [create   (amount/c . -> . account?)]
 [balance  (account? . -> . amount/c)]
 [withdraw (->i ([acc account?]
                 [amt (acc) (and/c amount/c (<=/c (balance acc)))])
                [result (acc amt)
                        (and/c account?
                               (lambda (res)
                                 (>= (balance res)
                                     (- (balance acc) amt))))])]
 [deposit  (->i ([acc account?]
                 [amt amount/c])
                [result (acc amt)
                        (and/c account?
                               (lambda (res)
                                 (>= (balance res)
```

```
                                        (+ (balance acc) amt)))))]]])

; section 3: the function definitions
(define balance account-balance)

(define (create amt) (account amt))

(define (withdraw a amt)
  (account (- (account-balance a) amt)))

(define (deposit a amt)
  (account (+ (account-balance a) amt)))
```

The contracts in section 2 provide typical type-like guarantees for create and balance. For withdraw and deposit, however, the contracts check and guarantee the more complicated constraints on balance and deposit. The contract on the second argument to withdraw uses (balance acc) to check whether the supplied withdrawal amount is small enough, where acc is the name given within ->i to the function's first argument. The contract on the result of withdraw uses both acc and amt to guarantee that no more than that requested amount was withdrawn. The contract on deposit similarly uses acc and amount in the result contract to guarantee that at least as much money as provided was deposited into the account.

As written above, when a contract check fails, the error message is not great. The following revision uses flat-named-contract within a helper function mk-account-contract to provide better error messages.

```
#lang racket

; section 1: the contract definitions
(struct account (balance))
(define amount/c natural-number/c)

(define msg> "account a with balance larger than ~a expected")
(define msg< "account a with balance less than ~a expected")

(define (mk-account-contract acc amt op msg)
  (define balance0 (balance acc))
  (define (ctr a)
    (and (account? a) (op balance0 (balance a))))
  (flat-named-contract (format msg balance0) ctr))

; section 2: the exports
(provide/contract
 [create   (amount/c . -> . account?)]
 [balance  (account? . -> . amount/c)]
```

```
   [withdraw (->i ([acc account?]
                    [amt (acc) (and/c amount/c (<=/c (balance acc)))])
                  [result (acc amt) (mk-account-
contract acc amt >= msg>)])]
   [deposit  (->i ([acc account?]
                    [amt amount/c])
                  [result (acc amt)
                          (mk-account-contract acc amt <= msg<)])])])

  ; section 3: the function definitions
  (define balance account-balance)

  (define (create amt) (account amt))

  (define (withdraw a amt)
    (account (- (account-balance a) amt)))

  (define (deposit a amt)
    (account (+ (account-balance a) amt)))
```

7.3.7 Checking State Changes

The `->i` contract combinator can also ensure that a function only modifies state according to certain constraints. For example, consider this contract (it is a slightly simplified from the function `preferences:add-panel` in the framework):

```
(->i ([parent (is-a?/c area-container-window<%>)])
     [_ (parent)
        (let ([old-children (send parent get-children)])
          (λ (child)
            (andmap eq?
                    (append old-children (list child))
                    (send parent get-children))))])
```

It says that the function accepts a single argument, named `parent`, and that `parent` must be an object matching the interface `area-container-window<%>`.

The range contract ensures that the function only modifies the children of `parent` by adding a new child to the front of the list. It accomplishes this by using the _ instead of a normal identifier, which tells the contract library that the range contract does not depend on the values of any of the results, and thus the contract library evaluates the expression following the _ when the function is called, instead of when it returns. Therefore the call to the `get-children` method happens before the function under the contract is called. When the function under contract returns, its result is passed in as `child`, and the contract ensures that the children after the function return are the same as the children before the function called,

but with one more child, at the front of the list.

To see the difference in a toy example that focuses on this point, consider this program

```
#lang racket
(define x '())
(define (get-x) x)
(define (f) (set! x (cons 'f x)))
(provide/contract
 [f (->i () [_ (begin (set! x (cons 'ctc x)) any/c)])]
 [get-x (-> (listof symbol?))])
```

If you were to require this module, call `f`, then the result of `get-x` would be `'(f ctc)`. In contrast, if the contract for `f` were

```
(->i () [res (begin (set! x (cons 'ctc x)) any/c)])
```

(only changing the underscore to `res`), then the result of `get-x` would be `'(ctc f)`.

7.3.8 Multiple Result Values

The function `split` consumes a list of `char`s and delivers the string that occurs before the first occurrence of `#\newline` (if any) and the rest of the list:

```
(define (split l)
  (define (split l w)
    (cond
      [(null? l) (values (list->string (reverse w)) '())]
      [(char=? #\newline (car l))
       (values (list->string (reverse w)) (cdr l))]
      [else (split (cdr l) (cons (car l) w))]))
  (split l '()))
```

It is a typical multiple-value function, returning two values by traversing a single list.

The contract for such a function can use the ordinary function arrow `->`, since `->` treats `values` specially when it appears as the last result:

```
(provide/contract
 [split (-> (listof char?)
            (values string? (listof char?)))])
```

The contract for such a function can also be written using `->*`:

```
(provide/contract
 [split (->* ((listof char?))
```

```
          ()
          (values string? (listof char?)))])
```

As before, the contract for the argument with ->* is wrapped in an extra pair of parentheses (and must always be wrapped like that) and the empty pair of parentheses indicates that there are no optional arguments. The contracts for the results are inside `values`: a string and a list of characters.

Now, suppose that we also want to ensure that the first result of `split` is a prefix of the given word in list format. In that case, we need to use the ->i contract combinator:

```
(define (substring-of? s)
  (flat-named-contract
    (format "substring of ~s" s)
    (lambda (s2)
      (and (string? s2)
           (<= (string-length s2) s)
           (equal? (substring s 0 (string-length s2)) s2)))))

(provide/contract
 [split (->i ([fl (listof char?)])
             (values [s (fl) (substring-of (list->string fl))]
                     [c (listof char?)]))])
```

Like ->*, the ->i combinator uses a function over the argument to create the range contracts. Yes, it doesn't just return one contract but as many as the function produces values: one contract per value. In this case, the second contract is the same as before, ensuring that the second result is a list of `char`s. In contrast, the first contract strengthens the old one so that the result is a prefix of the given word.

This contract is expensive to check, of course. Here is a slightly cheaper version:

```
(provide/contract
 [split (->i ([fl (listof char?)])
             (values [s (fl) (string-len/c (length fl))]
                     [c (listof char?)]))])
```

7.3.9 Fixed but Statically Unknown Arities

Imagine yourself writing a contract for a function that accepts some other function and a list of numbers that eventually applies the former to the latter. Unless the arity of the given function matches the length of the given list, your procedure is in trouble.

Consider this `n-step` function:

```
; (number ... -> (union #f number?)) (listof number) -> void
(define (n-step proc inits)
  (let ([inc (apply proc inits)])
    (when inc
      (n-step proc (map (λ (x) (+ x inc)) inits)))))
```

The argument of `n-step` is `proc`, a function `proc` whose results are either numbers or false, and a list. It then applies `proc` to the list `inits`. As long as `proc` returns a number, `n-step` treats that number as an increment for each of the numbers in `inits` and recurs. When `proc` returns `false`, the loop stops.

Here are two uses:

```
; nat -> nat
(define (f x)
  (printf "~s\n" x)
  (if (= x 0) #f -1))
(n-step f '(2))

; nat nat -> nat
(define (g x y)
  (define z (+ x y))
  (printf "~s\n" (list x y z))
  (if (= z 0) #f -1))

(n-step g '(1 1))
```

A contract for `n-step` must specify two aspects of `proc`'s behavior: its arity must include the number of elements in `inits`, and it must return either a number or `#f`. The latter is easy, the former is difficult. At first glance, this appears to suggest a contract that assigns a *variable-arity* to `proc`:

```
(->* ()
     (listof any/c)
     (or/c number? false/c))
```

This contract, however, says that the function must accept *any* number of arguments, not a *specific* but *undetermined* number. Thus, applying `n-step` to `(lambda (x) x)` and `(list 1)` breaks the contract because the given function accepts only one argument.

The correct contract uses the `unconstrained-domain->` combinator, which specifies only the range of a function, not its domain. It is then possible to combine this contract with an arity test to specify the correct `n-step`'s contract:

```
(provide/contract
 [n-step
  (->i ([proc (inits)
```

```
        (and/c (unconstrained-domain->
                  (or/c false/c number?))
                (λ (f) (procedure-arity-includes?
                          f
                          (length inits)))))]
   [inits (listof number?)])
  ()
  any)])
```

7.4   Contracts: A Thorough Example

This section develops several different flavors of contracts for one and the same example:
Racket's `argmax` function. According to its Racket documentation, the function consumes
a procedure `proc` and a non-empty list of values, `lst`. It

> returns the *first* element in the list `lst` that maximizes the result of `proc`.

The emphasis on *first* is ours.

Examples:

```
> (argmax add1 (list 1 2 3))
3
> (argmax sqrt (list 0.4 0.9 0.16))
0.9
> (argmax second '((a 2) (b 3) (c 4) (d 1) (e 4)))
'(c 4)
```

Here is the simplest possible contract for this function:

version 1

```
#lang racket

(define (argmax f lov) ...)

(provide/contract
  [argmax (-> (-> any/c real?) (and/c pair? list?) any/c)])
```

This contract captures two essential conditions of the informal description of `argmax`:

- the given function must produce numbers that are comparable according to `<`. In
  particular, the contract `(-> any/c number?)` would not do, because `number?` also
  recognizes complex numbers in Racket.

138

- the given list must contain at least one item.

When combined with the name, the contract explains the behavior of `argmax` at the same level as an ML function type in a module signature (except for the non-empty list aspect).

Contracts may communicate significantly more than a type signature, however. Take a look at this second contract for `argmax`:

version 2

```
#lang racket

(define (argmax f lov) ...)

(provide/contract
  [argmax
    (->i ([f (-> any/c real?)] [lov (and/c pair? list?)]) ()
          (r (f lov)
              (lambda (r)
                (define f@r (f r))
                (for/and ((v lov)) (>= f@r (f v))))))])
```

It is a *dependent* contract that names the two arguments and uses the names to impose a predicate on the result. This predicate computes `(f r)` – where `r` is the result of `argmax` – and then validates that this value is greater than or equal to all values of `f` on the items of `lov`.

Is it possible that `argmax` could cheat by returning a random value that accidentally maximizes `f` over all elements of `lov`? With a contract, it is possible to rule out this possibility:

version 2 rev. a

```
#lang racket

(define (argmax f lov) ...)

(provide/contract
  [argmax
    (->i ([f (-> any/c real?)] [lov (and/c pair? list?)]) ()
          (r (f lov)
              (lambda (r)
                (define f@r (f r))
                (and
                  (memq r lov)
                  (for/and ((v lov)) (>= f@r (f v)))))))])
```

The `memq` function ensures that `r` is *intensionally equal* to one of the members of `lov`. Of course, a moment's worth of reflection shows that it is impossible to make up such a value. That is, "pointer equality" for those who prefer to think at the hardware level.

139

Functions are opaque values in Racket and without applying a function, it is impossible to determine whether some random input value produces an output value or triggers some exception. So we ignore this possibility from here on.

Version 2 formulates the overall sentiment of `argmax`'s documentation, but it fails to bring across that the result is the *first* element of the given list that maximizes the given function `f`. Here is a version that communicates this second aspect of the informal documentation:

version 3

```
#lang racket

(define (argmax f lov) ...)

(provide/contract
  [argmax
    (->i ([f (-> any/c real?)] [lov (and/c pair? list?)]) ()
         (r (f lov)
            (lambda (r)
              (define f@r (f r))
              (and (for/and ((v lov)) (>= f@r (f v)))
                   (eq? (first (memf (lambda (v) (= (f v) f@r)) lov))
                        r)))))])
```

That is, the `memf` function determines the first element of `lov` whose value under `f` is equal to `r`'s value under `f`. If this element is intensionally equal to `r`, the result of `argmax` is correct.

This second refinement step introduces two problems. First, both conditions recompute the values of `f` for all elements of `lov`. Second, the contract is now quite difficult to read. Contracts should have a concise formulation that a client can comprehend with a simple scan. Let us eliminate the readability problem with two auxiliary functions that have reasonably meaningful names:

version 3 rev. a

```
#lang racket

(define (argmax f lov) ...)

(provide/contract
  [argmax
    (->i ([f (-> any/c real?)] [lov (and/c pair? list?)]) ()
         (r (f lov)
            (lambda (r)
              (define f@r (f r))
              (and (is-first-max? r f@r f lov)
                   (dominates-all f@r f lov)))))])
```

140

```
; where

; f@r is greater or equal to all (f v) for v in lov
(define (dominates-all f@r f lov)
  (for/and ((v lov)) (>= (f v) f@r)))

; r is eq? to the first element v of lov for which (pred? v)
(define (is-first-max? r f@r f lov)
  (eq? (first (memf (lambda (v) (= (f v) f@r)) lov)) r))
```

The names of the two predicates express their functionality and, in principle, render it unnecessary to read their definitions.

This step leaves us with the problem of the newly introduced inefficiency. To avoid the recomputation of (f v) for all v on lov, we change the contract so that it computes these values and reuses them as needed:

version 3 rev. b

```
#lang racket

(define (argmax f lov) ...)

(provide/contract
  [argmax
    (->i ([f (-> any/c real?)] [lov (and/c pair? list?)]) ()
         (r (f lov)
            (lambda (r)
              (define f@r (f r))
              (define flov (map f lov))
              (and (is-first-max? r f@r (map list lov flov))
                   (dominates-all f@r flov))))))])

; where

; f@r is greater or equal to all f@v in flov
(define (dominates-all f@r flov)
  (for/and ((f@v flov)) (>= f@r f@v)))

; r is (second x) for the first x in flov+lov s.t. (= (first x)
f@r)
(define (is-first-max? r f@r lov+flov)
  (define fst (first lov+flov))
  (if (= (second fst) f@r)
      (eq? (first fst) r)
      (is-first-max? f@r r (rest lov+flov))))
```

141

Now the predicate on the result once again computes all values of `f` for elements of `lov` once.

Version 3 may still be too eager when it comes to calling `f`. While Racket's `argmax` always calls `f` no matter how many items `lov` contains, let us imagine for illustrative purposes that our own implementation first checks whether the list is a singleton. If so, the first element would be the only element of `lov` and in that case there would be no need to compute `(f r)`. As a matter of fact, since `f` may diverge or raise an exception for some inputs, `argmax` should avoid calling `f` when possible.

The following contract demonstrates how a higher-order dependent contract needs to be adjusted so as to avoid being over-eager:

version 4

```racket
#lang racket

(define (argmax f lov)
  (if (empty? (rest lov))
      (first lov)
      ...))

(provide/contract
  [argmax
    (->i ([f (-> any/c real?)] [lov (and/c pair? list?)]) ()
         (r (f lov)
            (lambda (r)
              (cond
                [(empty? (rest lov)) (eq? (first lov) r)]
                [else
                 (define f@r (f r))
                 (define flov (map f lov))
                 (and (is-first-max? r f@r (map list lov flov))
                      (dominates-all f@r flov))]))))])

; where

; f@r is greater or equal to all f@v in flov
(define (dominates-all f@r lov) ...)

; r is (second x) for the first x in flov+lov s.t. (= (first x)
f@r)
(define (is-first-max? r f@r lov+flov) ...)
```

Note that such considerations don't apply to the world of first-order contracts. Only a higher-order (or lazy) language forces the programmer to express contracts with such precision.

The problem of diverging or exception-raising functions should alert the reader to the even more general problem of functions with side-effects. If the given function `f` has visible effects – say it logs its calls to a file – then the clients of `argmax` will be able to observe two sets of logs for each call to `argmax`. To be precise, if the list of values contains more than one element, the log will contain two calls of `f` per value on `lov`. If `f` is expensive to compute, doubling the calls imposes a high cost.

To avoid this cost and to signal problems with overly eager contracts, a contract system could record the i/o of contracted function arguments and use these hashtables in the dependency specification. This is a topic of on-going research in PLT. Stay tuned.

7.5 Contracts on Structures

Modules deal with structures in two ways. First they export `struct` definitions, i.e., the ability to create structs of a certain kind, to access their fields, to modify them, and to distinguish structs of this kind against every other kind of value in the world. Second, on occasion a module exports a specific struct and wishes to promise that its fields contain values of a certain kind. This section explains how to protect structs with contracts for both uses.

7.5.1 Guarantees for a Specific Value

If your module defines a variable to be a structure, then you can specify the structure's shape using `struct/c`:

```
#lang racket
(require lang/posn)

(define origin (make-posn 0 0))

(provide/contract
 [origin (struct/c posn zero? zero?)])
```

In this example, the module imports a library for representing positions, which exports a `posn` structure. One of the `posn`s it creates and exports stands for the origin, i.e., (0,0), of the grid.

See also `vector/c` and similar contract combinators for (flat) compound data.

7.5.2 Guarantees for All Values

The book *How to Design Programs* teaches that `posn`s should contain only numbers in their two fields. With contracts we would enforce this informal data definition as follows:

```
#lang racket
```

```
(struct posn (x y))

(provide/contract
 [struct posn ((x number?) (y number?))]
 [p-okay posn?]
 [p-sick posn?])

(define p-okay (posn 10 20))
(define p-sick (posn 'a 'b))
```

This module exports the entire structure definition: `posn`, `posn?`, `posn-x`, `posn-y`, `set-posn-x!`, and `set-posn-y!`. Each function enforces or promises that the two fields of a `posn` structure are numbers — when the values flow across the module boundary. Thus, if a client calls `posn` on `10` and `'a`, the contract system signals a contract violation.

The creation of `p-sick` inside of the `posn` module, however, does not violate the contracts. The function `posn` is used internally, so `'a` and `'b` don't cross the module boundary. Similarly, when `p-sick` crosses the boundary of `posn`, the contract promises a `posn?` and nothing else. In particular, this check does *not* require that the fields of `p-sick` are numbers.

The association of contract checking with module boundaries implies that `p-okay` and `p-sick` look alike from a client's perspective until the client extracts the pieces:

```
#lang racket
(require lang/posn)

... (posn-x p-sick) ...
```

Using `posn-x` is the only way the client can find out what a `posn` contains in the `x` field. The application of `posn-x` sends `p-sick` back into the `posn` module and the result value – `'a` here – back to the client, again across the module boundary. At this very point, the contract system discovers that a promise is broken. Specifically, `posn-x` doesn't return a number but a symbol and is therefore blamed.

This specific example shows that the explanation for a contract violation doesn't always pinpoint the source of the error. The good news is that the error is located in the `posn` module. The bad news is that the explanation is misleading. Although it is true that `posn-x` produced a symbol instead of a number, it is the fault of the programmer who created a `posn` from symbols, i.e., the programmer who added

```
(define p-sick (posn 'a 'b))
```

to the module. So, when you are looking for bugs based on contract violations, keep this example in mind.

If we want to fix the contract for `p-sick` so that the error is caught when `sick` is exported, a single change suffices:

```
(provide/contract
  ...
  [p-sick (struct/c posn number? number?)])
```

That is, instead of exporting `p-sick` as a plain `posn?`, we use a `struct/c` contract to
enforce constraints on its components.

7.5.3 Checking Properties of Data Structures

Contracts written using `struct/c` immediately check the fields of the data structure, but
sometimes this can have disastrous effects on the performance of a program that does not,
itself, inspect the entire data structure.

As an example, consider the binary search tree search algorithm. A binary search tree is like
a binary tree, except that the numbers are organized in the tree to make searching the tree
fast. In particular, for each interior node in the tree, all of the numbers in the left subtree are
smaller than the number in the node, and all of the numbers in the right subtree are larger
than the number in the node.

We can implement a search function `in?` that takes advantage of the structure of the binary
search tree.

```
#lang racket

(struct node (val left right))

; determines if 'n' is in the binary search tree 'b',
; exploiting the binary search tree invariant
(define (in? n b)
  (cond
    [(null? b) #f]
    [else (cond
            [(= n (node-val b))
             #t]
            [(< n (node-val b))
             (in? n (node-left b))]
            [(> n (node-val b))
             (in? n (node-right b))])]))

; a predicate that identifies binary search trees
(define (bst-between? b low high)
  (or (null? b)
      (and (<= low (node-val b) high)
           (bst-between? (node-left b) low (node-val b))
           (bst-between? (node-right b) (node-val b) high))))
```

```
(define (bst? b) (bst-between? b -inf.0 +inf.0))

(provide (struct node (val left right)))
(provide/contract
 [bst? (any/c . -> . boolean?)]
 [in? (number? bst? . -> . boolean?)])
```

In a full binary search tree, this means that the `in?` function only has to explore a logarithmic number of nodes.

The contract on `in?` guarantees that its input is a binary search tree. But a little careful thought reveals that this contract defeats the purpose of the binary search tree algorithm. In particular, consider the inner `cond` in the `in?` function. This is where the `in?` function gets its speed: it avoids searching an entire subtree at each recursive call. Now compare that to the `bst-between?` function. In the case that it returns `#t`, it traverses the entire tree, meaning that the speedup of `in?` is lost.

In order to fix that, we can employ a new strategy for checking the binary search tree contract. In particular, if we only checked the contract on the nodes that `in?` looks at, we can still guarantee that the tree is at least partially well-formed, but without changing the complexity.

To do that, we need to use `define-contract-struct` in place of `struct`. Like `struct` (and more like `define-struct`), `define-contract-struct` defines a maker, predicate, and selectors for a new structure. Unlike `define-struct`, it also defines contract combinators, in this case `node/c` and `node/dc`. Also unlike `define-struct`, it does not allow mutators, making its structs always immutable.

The `node/c` function accepts a contract for each field of the struct and returns a contract on the struct. More interestingly, the syntactic form `node/dc` allows us to write dependent contracts, i.e., contracts where some of the contracts on the fields depend on the values of other fields. We can use this to define the binary search tree contract:

```
#lang racket

(define-contract-struct node (val left right))

; determines if 'n' is in the binary search tree 'b'
(define (in? n b) ... as before ...)

; bst-between : number number -> contract
; builds a contract for binary search trees
; whose values are between low and high
(define (bst-between/c low high)
  (or/c null?
        (node/dc [val (between/c low high)]
                 [left (val) (bst-between/c low val)]
```

```
                      [right (val) (bst-between/c val high)]]))))

  (define bst/c (bst-between/c -inf.0 +inf.0))

  (provide make-node node-left node-right node-val node?)
  (provide/contract
   [bst/c contract?]
   [in? (number? bst/c . -> . boolean?)])
```

In general, each use of `node/dc` must name the fields and then specify contracts for each field. In the above, the `val` field is a contract that accepts values between `low` and `high`. The `left` and `right` fields are dependent on the value of the `val` field, indicated by their second sub-expressions. Their contracts are built by recursive calls to the `bst-between/c` function. Taken together, this contract ensures the same thing that the `bst-between?` function checked in the original example, but here the checking only happens as `in?` explores the tree.

Although this contract improves the performance of `in?`, restoring it to the logarithmic behavior that the contract-less version had, it is still imposes a fairly large constant overhead. So, the contract library also provides `define-opt/c` that brings down that constant factor by optimizing its body. Its shape is just like the `define` above. It expects its body to be a contract and then optimizes that contract.

```
  (define-opt/c (bst-between/c low high)
    (or/c null?
          (node/dc [val (between/c low high)]
                   [left (val) (bst-between/c low val)]
                   [right (val) (bst-between/c val high)]]))))
```

7.6   Abstract Contracts using `#:exists` and `#:∃`

The contract system provides existential contracts that can protect abstractions, ensuring that clients of your module cannot depend on the precise representation choices you make for your data structures.

The `provide/contract` form allows you to write

```
  #:∃ name-of-a-new-contract
```

as one of its clauses. This declaration introduces the variable *name-of-a-new-contract*, binding it to a new contract that hides information about the values it protects.

As an example, consider this (simple) implementation of a stack datastructure:

```
  #lang racket
```

You can type
`#:exists` instead
of `#:∃` if you
cannot easily type
unicode characters;
in DrRacket, typing
`\exists` followed
by either alt-\ or
control-\
(depending on your
platform) will
produce ∃.

147

```
(define empty '())
(define (enq top queue) (append queue (list top)))
(define (next queue) (car queue))
(define (deq queue) (cdr queue))
(define (empty? queue) (null? queue))

(provide/contract
 [empty (listof integer?)]
 [enq (-> integer? (listof integer?) (listof integer?))]
 [next (-> (listof integer?) integer?)]
 [deq (-> (listof integer?) (listof integer?))]
 [empty? (-> (listof integer?) boolean?)])
```

This code implements a queue purely in terms of lists, meaning that clients of this data structure might use `car` and `cdr` directly on the data structure (perhaps accidentally) and thus any change in the representation (say to a more efficient representation that supports amortized constant time enqueue and dequeue operations) might break client code.

To ensure that the stack representation is abstract, we can use `#:∃` in the `provide/contract` expression, like this:

```
(provide/contract
 #:∃ stack
 [empty stack]
 [enq (-> integer? stack stack)]
 [next (-> stack integer?)]
 [deq (-> stack (listof integer?))]
 [empty? (-> stack boolean?)])
```

Now, if clients of the data structure try to use `car` and `cdr`, they receive an error, rather than mucking about with the internals of the queues.

See also §7.8.2 "Exists Contracts and Predicates".

7.7 Additional Examples

This section illustrates the current state of Racket's contract implementation with a series of examples from *Design by Contract, by Example* [Mitchell02].

Mitchell and McKim's principles for design by contract DbC are derived from the 1970s style algebraic specifications. The overall goal of DbC is to specify the constructors of an algebra in terms of its observers. While we reformulate Mitchell and McKim's terminology and we use a mostly applicative approach, we retain their terminology of "classes" and "objects":

- **Separate queries from commands.**

  A *query* returns a result but does not change the observable properties of an object. A *command* changes the visible properties of an object, but does not return a result. In applicative implementation a command typically returns an new object of the same class.

- **Separate basic queries from derived queries.**

  A *derived query* returns a result that is computable in terms of basic queries.

- **For each derived query, write a post-condition contract that specifies the result in terms of the basic queries.**

- **For each command, write a post-condition contract that specifies the changes to the observable properties in terms of the basic queries.**

- **For each query and command, decide on suitable pre-condition contract.**

Each of the following sections corresponds to a chapter in Mitchell and McKim's book (but not all chapters show up here). We recommend that you read the contracts first (near the end of the first modules), then the implementation (in the first modules), and then the test module (at the end of each section).

Mitchell and McKim use Eiffel as the underlying programming language and employ a conventional imperative programming style. Our long-term goal is to transliterate their examples into applicative Racket, structure-oriented imperative Racket, and Racket's class system.

Note: To mimic Mitchell and McKim's informal notion of parametericity (parametric polymorphism), we use first-class contracts. At several places, this use of first-class contracts improves on Mitchell and McKim's design (see comments in interfaces).

7.7.1 A Customer-Manager Component

This first module contains some struct definitions in a separate module in order to better track bugs.

```
#lang racket
; data definitions

(define id? symbol?)
(define id-equal? eq?)
(define-struct basic-customer (id name address) #:mutable)

; interface
(provide/contract
```

```
 [id?                         (-> any/c boolean?)]
 [id-equal?                   (-> id? id? boolean?)]
 [struct basic-customer ((id id?)
                         (name string?)
                         (address string?))]])
; end of interface
```

This module contains the program that uses the above.

```
#lang racket

(require "1.ss") ; the module just above

; implementation
; [listof (list basic-customer? secret-info)]
(define all '())

(define (find c)
  (define (has-c-as-key p)
    (id-equal? (basic-customer-id (car p)) c))
  (define x (filter has-c-as-key all))
  (if (pair? x) (car x) x))

(define (active? c)
  (define f (find c))
  (pair? (find c)))

(define not-active? (compose not active? basic-customer-id))

(define count 0)

(define (add c)
  (set! all (cons (list c 'secret) all))
  (set! count (+ count 1)))

(define (name id)
  (define bc-with-id (find id))
  (basic-customer-name (car bc-with-id)))

(define (set-name id name)
  (define bc-with-id (find id))
  (set-basic-customer-name! (car bc-with-id) name))

(define c0 0)
; end of implementation
```

```
(provide/contract
 ; how many customers are in the db?
 [count     natural-number/c]
 ; is the customer with this id active?
 [active?  (-> id? boolean?)]
 ; what is the name of the customer with this id?
 [name      (-> (and/c id? active?) string?)]
 ; change the name of the customer with this id
 [set-name (->d ([id id?] [nn string?])
               ()
               [result any/c] ; result contract
               #:post-cond
               (string=? (name id) nn))]

 [add       (->d ([bc (and/c basic-customer? not-active?)])
               ()
               ; A pre-post condition contract must use
               ; a side-effect to express this contract
               ; via post-conditions
               #:pre-cond (set! c0 count)
               [result any/c] ; result contract
               #:post-cond (> count c0))])
```

The tests:

```
#lang racket
(require rackunit rackunit/text-ui "1.ss" "1b.ss")

(add (make-basic-customer 'mf "matthias" "brookstone"))
(add (make-basic-customer 'rf "robby" "beverly hills park"))
(add (make-basic-customer 'fl "matthew" "pepper clouds town"))
(add (make-basic-customer 'sk "shriram" "i city"))

(run-tests
 (test-suite
  "manager"
  (test-equal? "id lookup" "matthias" (name 'mf))
  (test-equal? "count" 4 count)
  (test-true "active?" (active? 'mf))
  (test-false "active? 2" (active? 'kk))
  (test-true "set name" (void? (set-name 'mf "matt")))))))
```

7.7.2 A Parameteric (Simple) Stack

```racket
#lang racket

; a contract utility
(define (eq/c x) (lambda (y) (eq? x y)))

(define-struct stack (list p? eq))

(define (initialize p? eq) (make-stack '() p? eq))
(define (push s x)
  (make-stack (cons x (stack-list s)) (stack-p? s) (stack-eq s)))
(define (item-at s i) (list-ref (reverse (stack-list s)) (- i 1)))
(define (count s) (length  (stack-list s)))
(define (is-empty? s) (null? (stack-list s)))
(define not-empty? (compose not is-empty?))
(define (pop s) (make-stack (cdr (stack-list s))
                            (stack-p? s)
                            (stack-eq s)))
(define (top s) (car (stack-list s)))

(provide/contract
 ; predicate
 [stack?      (-> any/c boolean?)]

 ; primitive queries
 ; how many items are on the stack?
 [count       (-> stack? natural-number/c)]

 ; which item is at the given position?
 [item-at
  (->d ([s stack?] [i (and/c positive? (<=/c (count s)))])
       ()
       [result (stack-p? s)])]

 ; derived queries
 ; is the stack empty?
 [is-empty?
  (->d ([s stack?])
       ()
       [result (eq/c (= (count s) 0))])]

 ; which item is at the top of the stack
 [top
  (->d ([s (and/c stack? not-empty?)])
```

```
      ()
      [t (stack-p? s)]  ; a stack item, t is its name
      #:post-cond
      ([stack-eq s] t (item-at s (count s))))]

 ; creation
 [initialize
  (->d ([p contract?] [s (p p . -> . boolean?)])
       ()
       ; Mitchell and McKim use (= (count s) 0) here to express
       ; the post-condition in terms of a primitive query
       [result (and/c stack? is-empty?)])]

 ; commands
 ; add an item to the top of the stack
 [push
  (->d ([s stack?] [x (stack-p? s)])
       ()
       [sn stack?]  ; result kind
       #:post-cond
       (and (= (+ (count s) 1) (count sn))
            ([stack-eq s] x (top sn))))]

 ; remove the item at the top of the stack
 [pop
  (->d ([s (and/c stack? not-empty?)])
       ()
       [sn stack?]  ; result kind
       #:post-cond
       (= (- (count s) 1) (count sn)))])
```

The tests:

```
#lang racket
(require rackunit rackunit/text-ui "2.ss")

(define s0 (initialize (flat-contract integer?) =))
(define s2 (push (push s0 2) 1))

(run-tests
 (test-suite
  "stack"
  (test-true
   "empty"
   (is-empty? (initialize (flat-contract integer?) =)))
  (test-true "push" (stack? s2))
```

```
     (test-true
      "push exn"
      (with-handlers ([exn:fail:contract? (lambda _ #t)])
        (push (initialize (flat-contract integer?)) 'a)
        #f))
     (test-true "pop" (stack? (pop s2)))
     (test-equal? "top" (top s2) 1)
     (test-equal? "toppop" (top (pop s2)) 2)))
```

7.7.3 A Dictionary

```
#lang racket

; a shorthand for use below
(define-syntax ⇒
  (syntax-rules ()
    [(⇒ antecedent consequent) (if antecedent consequent #t)]))

; implementation
(define-struct dictionary (l value? eq?))
; the keys should probably be another parameter (exercise)

(define (initialize p eq) (make-dictionary '() p eq))
(define (put d k v)
  (make-dictionary (cons (cons k v) (dictionary-l d))
                   (dictionary-value? d)
                   (dictionary-eq? d)))
(define (rem d k)
  (make-dictionary
   (let loop ([l (dictionary-l d)])
     (cond
       [(null? l) l]
       [(eq? (caar l) k) (loop (cdr l))]
       [else (cons (car l) (loop (cdr l)))]))
   (dictionary-value? d)
   (dictionary-eq? d)))
(define (count d) (length (dictionary-l d)))
(define (value-for d k) (cdr (assq k (dictionary-l d))))
(define (has? d k) (pair? (assq k (dictionary-l d))))
(define (not-has? d) (lambda (k) (not (has? d k))))
; end of implementation

; interface
(provide/contract
```

154

```scheme
; predicates
[dictionary? (-> any/c boolean?)]
; basic queries
; how many items are in the dictionary?
[count       (-> dictionary? natural-number/c)]
; does the dictionary define key k?
[has?         (->d ([d dictionary?] [k symbol?])
                   ()
                   [result boolean?]
                   #:post-cond
                   ((zero? (count d)) . ⇒ . (not result)))]
; what is the value of key k in this dictionary?
[value-for   (->d ([d dictionary?]
                    [k (and/c symbol? (lambda (k) (has? d k)))])
                   ()
                   [result (dictionary-value? d)])]
; initialization
; post condition: for all k in symbol, (has? d k) is false.
[initialize  (->d ([p contract?] [eq (p p . -> . boolean?)])
                   ()
                   [result (and/c dictionary? (compose zero? count))])]
; commands
; Mitchell and McKim say that put shouldn't consume Void (null
ptr)
; for v. We allow the client to specify a contract for all values
; via initialize. We could do the same via a key? parameter
; (exercise). add key k with value v to this dictionary
[put          (->d ([d dictionary?]
                    [k (and symbol? (not-has? d))]
                    [v (dictionary-value? d)])
                   ()
                   [result dictionary?]
                   #:post-cond
                   (and (has? result k)
                        (= (count d) (- (count result) 1))
                        ([dictionary-eq? d] (value-
for result k) v)))]
; remove key k from this dictionary
[rem          (->d ([d dictionary?]
                    [k (and/c symbol? (lambda (k) (has? d k)))])
                   ()
                   [result (and/c dictionary? not-has?)]
                   #:post-cond
                   (= (count d) (+ (count result) 1)))])
; end of interface
```

155

The tests:

```racket
#lang racket
(require rackunit rackunit/text-ui "3.ss")

(define d0 (initialize (flat-contract integer?) =))
(define d (put (put (put d0 'a 2) 'b 2) 'c 1))

(run-tests
 (test-suite
  "dictionaries"
  (test-equal? "value for" 2 (value-for d 'b))
  (test-false "has?" (has? (rem d 'b) 'b))
  (test-equal? "count" 3 (count d))))
```

7.7.4 A Queue

```racket
#lang racket

; Note: this queue doesn't implement the capacity restriction
; of Mitchell and McKim's queue but this is easy to add.

; a contract utility
(define (all-but-last l) (reverse (cdr (reverse l))))
(define (eq/c x) (lambda (y) (eq? x y)))

; implementation
(define-struct queue (list p? eq))

(define (initialize p? eq) (make-queue '() p? eq))
(define items queue-list)
(define (put q x)
  (make-queue (append (queue-list q) (list x))
              (queue-p? q)
              (queue-eq q)))
(define (count s) (length  (queue-list s)))
(define (is-empty? s) (null? (queue-list s)))
(define not-empty? (compose not is-empty?))
(define (rem s)
  (make-queue (cdr (queue-list s))
              (queue-p? s)
              (queue-eq s)))
(define (head s) (car (queue-list s)))
```

156

```scheme
; interface
(provide/contract
 ; predicate
 [queue?      (-> any/c boolean?)]

 ; primitive queries
 ; Imagine providing this 'query' for the interface of the module
 ; only. Then in Racket there is no reason to have count or is-
empty?
 ; around (other than providing it to clients). After all items is
 ; exactly as cheap as count.
 [items       (->d ([q queue?]) () [result (listof (queue-p? q))]])]

 ; derived queries
 [count       (->d ([q queue?])
                   ; We could express this second part of the post
                   ; condition even if count were a module "at-
tribute"
                   ; in the language of Eiffel; indeed it would use
the
                   ; exact same syntax (minus the arrow and domain).
                   ()
                   [result (and/c natural-number/c
                                  (=/c (length (items q))))]])]

 [is-empty?  (->d ([q queue?])
                  ()
                  [result (and/c boolean?
                                 (eq/c (null? (items q))))]])]

 [head       (->d ([q (and/c queue? (compose not is-empty?))])
                  ()
                  [result (and/c (queue-p? q)
                                 (eq/c (car (items q))))]])]
 ; creation
 [initialize (-> contract?
                 (contract? contract? . -> . boolean?)
                 (and/c queue? (compose null? items)))]

 ; commands
 [put        (->d ([oldq queue?] [i (queue-p? oldq)])
                  ()
                  [result
                   (and/c
                    queue?
                    (lambda (q)
```

```
                        (define old-items (items oldq))
                        (equal? (items q) (append old-
items (list i)))))]))]

  [rem        (->d ([oldq (and/c queue? (compose not is-empty?))])
                   ()
                   [result
                    (and/c queue?
                           (lambda (q)
                             (equal? (cdr (items oldq)) (items q))))])])
  ; end of interface
```

The tests:

```
#lang racket
(require rackunit rackunit/text-ui "5.ss")

(define s (put (put (initialize (flat-contract integer?) =) 2) 1))

(run-tests
 (test-suite
  "queue"
  (test-true
   "empty"
   (is-empty? (initialize (flat-contract integer?) =)))
  (test-true "put" (queue? s))
  (test-equal? "count" 2 (count s))
  (test-true "put exn"
             (with-handlers ([exn:fail:contract? (lambda _ #t)])
               (put (initialize (flat-contract integer?)) 'a)
               #f))
  (test-true "remove" (queue? (rem s)))
  (test-equal? "head" 2 (head s))))
```

7.8   Gotchas

7.8.1   Contracts and eq?

As a general rule, adding a contract to a program should either leave the behavior of the program unchanged, or should signal a contract violation. And this is almost true for Racket contracts, with one exception: eq?.

The eq? procedure is designed to be fast and does not provide much in the way of guarantees, except that if it returns true, it means that the two values behave identically in all respects.

158

Internally, this is implemented as pointer equality at a low-level so it exposes information about how Racket is implemented (and how contracts are implemented).

Contracts interact poorly with `eq?` because function contract checking is implemented internally as wrapper functions. For example, consider this module:

```racket
#lang racket

(define (make-adder x)
  (if (= 1 x)
      add1
      (lambda (y) (+ x y))))
(provide/contract [make-adder (-> number? (-> number? number?))])
```

It exports the `make-adder` function that is the usual curried addition function, except that it returns Racket's `add1` when its input is `1`.

You might expect that

```racket
(eq? (make-adder 1)
     (make-adder 1))
```

would return `#t`, but it does not. If the contract were changed to `any/c` (or even `(-> number? any/c)`), then the `eq?` call would return `#t`.

Moral: Do not use `eq?` on values that have contracts.

7.8.2 Exists Contracts and Predicates

Much like the `eq?` example above, `#:∃` contracts can change the behavior of a program.

Specifically, the `null?` predicate (and many other predicates) return `#f` for `#:∃` contracts, and changing one of those contracts to `any/c` means that `null?` might now return `#t` instead, resulting in arbitrarily different behavior depending on this boolean might flow around in the program.

```racket
#lang racket/exists
```

To work around the above problem, the `racket/exists` library behaves just like the `racket`, but where predicates signal errors when given `#:∃` contracts.

Moral: Do not use predicates on `#:∃` contracts, but if you're not sure, use `racket/exists` to be safe.

7.8.3 Defining Recursive Contracts

When defining a self-referential contract, it is natural to use `define`. For example, one might try to write a contract on streams like this:

```
> (define stream/c
    (promise/c
     (or/c
      null?
      (cons/c number? stream/c))))
reference to undefined identifier: stream/c
```

Unfortunately, this does not work because the value of `stream/c` is needed before it is defined. Put another way, all of the combinators evaluate their arguments eagerly, even thought the values that they accept do not.

Instead, use

```
(define stream/c
 (promise/c
  (or/c
   null?
   (cons/c 1
           (recursive-contract stream/c)))))
```

The use of `recursive-contract` delays the evaluation of the identifier `stream/c` until after the contract is first checked, long enough to ensure that `stream/c` is defined.

See also §7.5.3 "Checking Properties of Data Structures".

7.8.4 Mixing `set!` and `provide/contract`

The contract library assumes that variables exported via `provide/contract` are not assigned to, but does not enforce it. Accordingly, if you try to `set!` those variables, you may be surprised. Consider the following example:

```
> (module server racket
    (define (inc-x!) (set! x (+ x 1)))
    (define x 0)
    (provide/contract [inc-x! (-> void?)]
                      [x integer?]))
> (module client racket
    (require 'server)

    (define (print-latest) (printf "x is ~s\n" x))
```

```
      (print-latest)
      (inc-x!)
      (print-latest))
> (require 'client)
x is 0
x is 0
```

Both calls to `print-latest` print 0, even though the value of x has been incremented (and the change is visible inside the module x).

To work around this, export accessor functions, rather than exporting the variable directly, like this:

```
#lang racket

(define (get-x) x)
(define (inc-x!) (set! x (+ x 1)))
(define x 0)
(provide/contract [inc-x! (-> void?)]
                  [get-x (-> integer?)])
```

Moral: This is a bug that we will address in a future release.

8 Input and Output

A Racket *port* represents an input or output stream, such as a file, a terminal, a TCP connection, or an in-memory string. More specifically, an *input port* represents a stream from which a program can read data, and an *output port* represents a stream for writing data.

8.1 Varieties of Ports

Various functions create various kinds of ports. Here are a few examples:

- **Files:** The `open-output-file` function opens a file for writing, and `open-input-file` opens a file for reading.

  Examples:
  ```
  > (define out (open-output-file "data"))
  > (display "hello" out)
  > (close-output-port out)
  > (define in (open-input-file "data"))
  > (read-line in)
  "hello"
  > (close-input-port in)
  ```

  If a file exists already, then `open-output-file` raises an exception by default. Supply an option like `#:exists 'truncate` or `#:exists 'update` to re-write or update the file:

  Examples:
  ```
  > (define out (open-output-file "data" #:exists 'truncate))
  > (display "howdy" out)
  > (close-output-port out)
  ```

  Instead of having to match `open-input-file` and `open-output-file` calls, most Racket programmers will instead use `call-with-output-file`, which takes a function to call with the output port; when the function returns, the port is closed.

  Examples:
  ```
  > (call-with-output-file "data"
                           #:exists 'truncate
                           (lambda (out)
                             (display "hello" out)))
  > (call-with-input-file "data"
                          (lambda (in)
                            (read-line in)))
  "hello"
  ```

- **Strings:** The `open-output-string` function creates a port that accumulates data

162

into a string, and `get-output-string` extracts the accumulated string. The `open-input-string` function creates a port to read from a string.

Examples:

```
> (define p (open-output-string))
> (display "hello" p)
> (get-output-string p)
"hello"
> (read-line (open-input-string "goodbye\nfarewell"))
"goodbye"
```

- **TCP Connections:** The `tcp-connect` function creates both an input port and an output port for the client side of a TCP communication. The `tcp-listen` function creates a server, which accepts connections via `tcp-accept`.

Examples:

```
> (define server (tcp-listen 12345))
> (define-values (c-in c-out) (tcp-connect "localhost" 12345))
> (define-values (s-in s-out) (tcp-accept server))
> (display "hello\n" c-out)
> (close-output-port c-out)
> (read-line s-in)
"hello"
> (read-line s-in)
#<eof>
```

- **Process Pipes:** The `subprocess` function runs a new process at the OS level and returns ports that correspond to the subprocess's stdin, stdout, and stderr. (The first three arguments can be certain kinds of existing ports to connect directly to the subprocess, instead of creating new ports.)

Examples:

```
> (define-values (p stdout stdin stderr)
    (subprocess #f #f #f "/usr/bin/wc" "-w"))
> (display "a b c\n" stdin)
> (close-output-port stdin)
> (read-line stdout)
"       3"
> (close-input-port stdout)
> (close-input-port stderr)
```

- **Internal Pipes:** The `make-pipe` function returns two ports that are ends of a pipe. This kind of pipe is internal to Racket, and not related to OS-level pipes for communicating between different processes.

Examples:

```
> (define-values (in out) (make-pipe))
> (display "garbage" out)
> (close-output-port out)
> (read-line in)
```

```
    "garbage"
```

8.2 Default Ports

For most simple I/O functions, the target port is an optional argument, and the default is the *current input port* or *current output port*. Furthermore, error messages are written to the *current error port*, which is an output port. The `current-input-port`, `current-output-port`, and `current-error-port` functions return the corresponding current ports.

Examples:
```
> (display "Hi")
Hi
> (display "Hi" (current-output-port)) ; the same
Hi
```

If you start the `racket` program in a terminal, then the current input, output, and error ports are all connected to the terminal. More generally, they are connected to the OS-level stdin, stdout, and stderr. In this guide, the examples show output written to stdout in purple, and output written to stderr in red italics.

Examples:
```
(define (swing-hammer)
  (display "Ouch!" (current-error-port)))
> (swing-hammer)
Ouch!
```

The current-port functions are actually parameters, which means that their values can be set with `parameterize`.

Example:
```
> (let ([s (open-output-string)])
    (parameterize ([current-error-port s])
      (swing-hammer)
      (swing-hammer)
      (swing-hammer))
    (get-output-string s))
"Ouch!Ouch!Ouch!"
```

See §4.13 "Dynamic Binding: `parameterize`" for an introduction to parameters.

8.3 Reading and Writing Racket Data

As noted throughout §3 "Built-In Datatypes", Racket provides three ways to print an instance of a built-in value:

- `print`, which prints a value in the same way that is it printed for a REPL result; and

- `write`, which prints a value in such a way that `read` on the output produces the value back; and

- `display`, which tends to reduce a value to just its character or byte content—at least for those datatypes that are primarily about characters or bytes, otherwise it falls back to the same output as `write`.

Here are some examples using each:

```
> (print 1/2)          > (write 1/2)          > (display 1/2)
1/2                     1/2                    1/2
> (print #\x)          > (write #\x)          > (display #\x)
#\x                     #\x                    x
> (print "hello")      > (write "hello")      > (display "hello")
"hello"                 "hello"                hello
> (print #"goodbye")   > (write #"goodbye")   > (display #"goodbye")
#"goodbye"              #"goodbye"             goodbye
> (print '|pea pod|)   > (write '|pea pod|)   > (display '|pea pod|)
'|pea pod|              |pea pod|              pea pod
> (print '("i" pod))   > (write '("i" pod))   > (display '("i" pod))
'("i" pod)             ("i" pod)              (i pod)
> (print write)        > (write write)        > (display write)
#<procedure:write>     #<procedure:write>     #<procedure:write>
```

Overall, `print` as corresponds to the expression layer of Racket syntax, `write` corresponds to the reader layer, and `display` roughly corresponds to the character layer.

The `printf` function supports simple formatting of data and text. In the format string supplied to `printf`, `~a` `display`s the next argument, `~s` `write`s the next argument, and `~v` `print`s the next argument.

Examples:
```
(define (deliver who when what)
  (printf "Items ~a for shopper ~s: ~v" who when what))
> (deliver '("list") '("John") '("milk"))
Items (list) for shopper ("John"): '("milk")
```

After using `write`, as opposed to `display` or `print`, many forms of data can be read back in using `read`. The same values `print`ed can also be parsed by `read`, but the result may have extra quote forms, since a `print`ed form is meant to be read like an expression.

Examples:
```
> (define-values (in out) (make-pipe))
> (write "hello" out)
> (read in)
```

165

```
"hello"
> (write '("alphabet" soup) out)
> (read in)
'("alphabet" soup)
> (write #hash((a . "apple") (b . "banana")) out)
> (read in)
'#hash((b . "banana") (a . "apple"))
> (write '("alphabet" soup) out)
> (read in)
'("alphabet" soup)
```

8.4 Datatypes and Serialization

Prefab structure types (see §5.7 "Prefab Structure Types") automatically support *serialization*: they can be written to an output stream, and a copy can be read back in from an input stream:

```
> (define-values (in out) (make-pipe))
> (write #s(sprout bean) out)
> (read in)
'#s(sprout bean)
```

Other structure types created by struct, which offer more abstraction than prefab structure types, normally write either using #<....> notation (for opaque structure types) or using #(....) vector notation (for transparent structure types). In neither can can the result be read back in as an instance of the structure type:

```
> (struct posn (x y))
> (write (posn 1 2))
#<posn>
> (define-values (in out) (make-pipe))
> (write (posn 1 2) out)
> (read in)
UNKNOWN::0: read: bad syntax '#<'

> (struct posn (x y) #:transparent)
> (write (posn 1 2))
#(struct:posn 1 2)
> (define-values (in out) (make-pipe))
> (write (posn 1 2) out)
> (define v (read in))
> v
'#(struct:posn 1 2)
> (posn? v)
#f
```

166

```
> (vector? v)
#t
```

The `serializable-struct` form defines a structure type that can be `serialize`d to a value that can be printed using `write` and restored via `read`. The `serialize`d result can be `deserialize`d to get back an instance of the original structure type. The serialization form and functions are provided by the `racket/serialize` library.

Examples:
```
> (require racket/serialize)
> (serializable-struct posn (x y) #:transparent)
> (deserialize (serialize (posn 1 2)))
(posn 1 2)
> (write (serialize (posn 1 2)))
((2) 1 ((#f . deserialize-info:posn-v0)) 0 () () (0 1 2))
> (define-values (in out) (make-pipe))
> (write (serialize (posn 1 2)) out)
> (deserialize (read in))
(posn 1 2)
```

In addition to the names bound by `struct`, `serializable-struct` binds an identifier with deserialization information, and it automatically `provides` the deserialization identifier from a module context. This deserialization identifier is accessed reflectively when a value is deserialized.

8.5   Bytes, Characters, and Encodings

Functions like `read-line`, `read`, `display`, and `write` all work in terms of characters (which correspond to Unicode scalar values). Conceptually, they are implemented in terms of `read-char` and `write-char`.

More primitively, ports read and write bytes, instead of characters. The functions `read-byte` and `write-byte` read and write raw bytes. Other functions, such as `read-bytes-line`, build on top of byte operations instead of character operations.

In fact, the `read-char` and `write-char` functions are conceptually implemented in terms of `read-byte` and `write-byte`. When a single byte's value is less than 128, then it corresponds to an ASCII character. Any other byte is treated as part of a UTF-8 sequence, where UTF-8 is a particular standard way of encoding Unicode scalar values in bytes (which has the nice property that ASCII characters are encoded as themselves). Thus, a single `read-char` may call `read-byte` multiple times, and a single `write-char` may generate multiple output bytes.

The `read-char` and `write-char` operations *always* use a UTF-8 encoding. If you have a text stream that uses a different encoding, or if you want to generate a text stream in a differ-

167

ent encoding, use `reencode-input-port` or `reencode-output-port`. The `reencode-input-port` function converts an input stream from an encoding that you specify into a UTF-8 stream; that way, `read-char` sees UTF-8 encodings, even though the original used a different encoding. Beware, however, that `read-byte` also sees the re-encoded data, instead of the original byte stream.

8.6   I/O Patterns

If you want to process individual lines of a file, then you can use `for` with `in-lines`:

```
> (define (upcase-all in)
    (for ([l (in-lines in)])
      (display (string-upcase l))
      (newline)))
> (upcase-all (open-input-string
               (string-append
                "Hello, World!\n"
                "Can you hear me, now?")))
HELLO, WORLD!
CAN YOU HEAR ME, NOW?
```

If you want to determine whether "hello" appears in a file, then you could search separate lines, but it's even easier to simply apply a regular expression (see §9 "Regular Expressions") to the stream:

```
> (define (has-hello? in)
    (regexp-match? #rx"hello" in))
> (has-hello? (open-input-string "hello"))
#t
> (has-hello? (open-input-string "goodbye"))
#f
```

If you want to copy one port into another, use `copy-port` from `racket/port`, which efficiently transfers large blocks when lots of data is available, but also transfers small blocks immediately if that's all that is available:

```
> (define o (open-output-string))
> (copy-port (open-input-string "broom") o)
> (get-output-string o)
"broom"
```

9 Regular Expressions

A *regexp* value encapsulates a pattern that is described by a string or byte string. The regexp matcher tries to match this pattern against (a portion of) another string or byte string, which we will call the *text string*, when you call functions like `regexp-match`. The text string is treated as raw text, and not as a pattern.

9.1 Writing Regexp Patterns

A string or byte string can be used directly as a regexp pattern, or it can be prefixed with `#rx` to form a literal regexp value. For example, `#rx"abc"` is a string-based regexp value, and `#rx#"abc"` is a byte string-based regexp value. Alternately, a string or byte string can be prefixed with `#px`, as in `#px"abc"`, for a slightly extended syntax of patterns within the string.

Most of the characters in a regexp pattern are meant to match occurrences of themselves in the text string. Thus, the pattern `#rx"abc"` matches a string that contains the characters `a`, `b`, and `c` in succession. Other characters act as *metacharacters*, and some character sequences act as *metasequences*. That is, they specify something other than their literal selves. For example, in the pattern `#rx"a.c"`, the characters `a` and `c` stand for themselves, but the metacharacter `.` can match *any* character. Therefore, the pattern `#rx"a.c"` matches an `a`, any character, and `c` in succession.

If we needed to match the character `.` itself, we can escape it by precede it with a `\`. The character sequence `\.` is thus a metasequence, since it doesn't match itself but rather just `.`. So, to match `a`, `.`, and `c` in succession, we use the regexp pattern `#rx"a\\.c"`; the double `\` is an artifact of Racket strings, not the regexp pattern itself.

The `regexp` function takes a string or byte string and produces a regexp value. Use `regexp` when you construct a pattern to be matched against multiple strings, since a pattern is compiled to a regexp value before it can be used in a match. The `pregexp` function is like `regexp`, but using the extended syntax. Regexp values as literals with `#rx` or `#px` are compiled once and for all when they are read.

The `regexp-quote` function takes an arbitrary string and returns a string for a pattern that matches exactly the original string. In particular, characters in the input string that could serve as regexp metacharacters are escaped with a backslash, so that they safely match only themselves.

```
> (regexp-quote "cons")
"cons"
> (regexp-quote "list?")
"list\\?"
```

The `regexp-quote` function is useful when building a composite regexp from a mix of regexp strings and verbatim strings.

9.2 Matching Regexp Patterns

The `regexp-match-positions` function takes a regexp pattern and a text string, and it returns a match if the regexp matches (some part of) the text string, or `#f` if the regexp did not match the string. A successful match produces a list of *index pairs*.

Examples:
```
> (regexp-match-positions #rx"brain" "bird")
#f
> (regexp-match-positions #rx"needle" "hay needle stack")
'((4 . 10))
```

In the second example, the integers 4 and 10 identify the substring that was matched. The 4 is the starting (inclusive) index, and 10 the ending (exclusive) index of the matching substring:

```
> (substring "hay needle stack" 4 10)
"needle"
```

In this first example, `regexp-match-positions`'s return list contains only one index pair, and that pair represents the entire substring matched by the regexp. When we discuss sub-patterns later, we will see how a single match operation can yield a list of submatches.

The `regexp-match-positions` function takes optional third and fourth arguments that specify the indices of the text string within which the matching should take place.

```
> (regexp-match-positions
   #rx"needle"
   "his needle stack -- my needle stack -- her needle stack"
   20 39)
'((23 . 29))
```

Note that the returned indices are still reckoned relative to the full text string.

The `regexp-match` function is like `regexp-match-positions`, but instead of returning index pairs, it returns the matching substrings:

```
> (regexp-match #rx"brain" "bird")
#f
> (regexp-match #rx"needle" "hay needle stack")
'("needle")
```

When `regexp-match` is used with byte-string regexp, the result is a matching byte substring:

```
> (regexp-match #rx#"needle" #"hay needle stack")
'(#"needle")
```

If you have data that is in a port, there's no need to first read it into a string. Functions like `regexp-match` can match on the port directly:

```
> (define-values (i o) (make-pipe))
> (write "hay needle stack" o)
> (close-output-port o)
> (regexp-match #rx#"needle" i)
'(#"needle")
```

The `regexp-match?` function is like `regexp-match-positions`, but simply returns a boolean indicating whether the match succeeded:

```
> (regexp-match? #rx"brain" "bird")
#f
> (regexp-match? #rx"needle" "hay needle stack")
#t
```

The `regexp-split` function takes two arguments, a regexp pattern and a text string, and it returns a list of substrings of the text string; the pattern identifies the delimiter separating the substrings.

```
> (regexp-split #rx":" "/bin:/usr/bin:/usr/bin/X11:/usr/local/bin")
'("/bin" "/usr/bin" "/usr/bin/X11" "/usr/local/bin")
> (regexp-split #rx" " "pea soup")
'("pea" "soup")
```

If the first argument matches empty strings, then the list of all the single-character substrings is returned.

```
> (regexp-split #rx"" "smithereens")
'("" "s" "m" "i" "t" "h" "e" "r" "e" "e" "n" "s" "")
```

Thus, to identify one-or-more spaces as the delimiter, take care to use the regexp `#rx" +"`, not `#rx" *"`.

```
> (regexp-split #rx" +" "split pea     soup")
'("split" "pea" "soup")
> (regexp-split #rx" *" "split pea     soup")
'("" "s" "p" "l" "i" "t" "" "p" "e" "a" "" "s" "o" "u" "p" "")
```

The `regexp-replace` function replaces the matched portion of the text string by another string. The first argument is the pattern, the second the text string, and the third is either the string to be inserted or a procedure to convert matches to the insert string.

```
> (regexp-replace #rx"te" "liberte" "ty")
```

171

```
"liberty"
> (regexp-replace #rx"." "racket" string-upcase)
"Racket"
```

If the pattern doesn't occur in the text string, the returned string is identical to the text string.

The `regexp-replace*` function replaces *all* matches in the text string by the insert string:

```
> (regexp-replace* #rx"te" "liberte egalite fraternite" "ty")
"liberty egality fratyrnity"
> (regexp-replace* #rx"[ds]" "drracket" string-upcase)
"Drracket"
```

9.3   Basic Assertions

The *assertions* `^` and `$` identify the beginning and the end of the text string, respectively. They ensure that their adjoining regexps match at one or other end of the text string:

```
> (regexp-match-positions #rx"^contact" "first contact")
#f
```

The regexp above fails to match because `contact` does not occur at the beginning of the text string. In

```
> (regexp-match-positions #rx"laugh$" "laugh laugh laugh laugh")
'((18 . 23))
```

the regexp matches the *last* `laugh`.

The metasequence `\b` asserts that a word boundary exists, but this metasequence works only with `#px` syntax. In

```
> (regexp-match-positions #px"yack\\b" "yackety yack")
'((8 . 12))
```

the `yack` in `yackety` doesn't end at a word boundary so it isn't matched. The second `yack` does and is.

The metasequence `\B` (also `#px` only) has the opposite effect to `\b`; it asserts that a word boundary does not exist. In

```
> (regexp-match-positions #px"an\\B" "an analysis")
'((3 . 5))
```

the `an` that doesn't end in a word boundary is matched.

9.4   Characters and Character Classes

Typically, a character in the regexp matches the same character in the text string. Sometimes it is necessary or convenient to use a regexp metasequence to refer to a single character. For example, the metasequence `\.` matches the period character.

The metacharacter `.` matches *any* character (other than newline in multi-line mode; see §9.6.3 "Cloisters"):

```
> (regexp-match #rx"p.t" "pet")
'("pet")
```

The above pattern also matches `pat`, `pit`, `pot`, `put`, and `p8t`, but not `peat` or `pfffft`.

A *character class* matches any one character from a set of characters. A typical format for this is the *bracketed character class* `[...]`, which matches any one character from the non-empty sequence of characters enclosed within the brackets. Thus, `#rx"p[aeiou]t"` matches `pat`, `pet`, `pit`, `pot`, `put`, and nothing else.

Inside the brackets, a `-` between two characters specifies the Unicode range between the characters. For example, `#rx"ta[b-dgn-p]"` matches `tab`, `tac`, `tad`, `tag`, `tan`, `tao`, and `tap`.

An initial `^` after the left bracket inverts the set specified by the rest of the contents; i.e., it specifies the set of characters *other than* those identified in the brackets. For example, `#rx"do[^g]"` matches all three-character sequences starting with `do` except `dog`.

Note that the metacharacter `^` inside brackets means something quite different from what it means outside. Most other metacharacters (`.`, `*`, `+`, `?`, etc.) cease to be metacharacters when inside brackets, although you may still escape them for peace of mind. A `-` is a metacharacter only when it's inside brackets, and when it is neither the first nor the last character between the brackets.

Bracketed character classes cannot contain other bracketed character classes (although they contain certain other types of character classes; see below). Thus, a `[` inside a bracketed character class doesn't have to be a metacharacter; it can stand for itself. For example, `#rx"a[b]"` matches `a`, `[`, and `b`.

Furthermore, since empty bracketed character classes are disallowed, a `]` immediately occurring after the opening left bracket also doesn't need to be a metacharacter. For example, `#rx"[]ab]"` matches `]`, `a`, and `b`.

9.4.1 Some Frequently Used Character Classes

In `#px` syntax, some standard character classes can be conveniently represented as metasequences instead of as explicit bracketed expressions: `\d` matches a digit (the same as `[0-9]`); `\s` matches an ASCII whitespace character; and `\w` matches a character that could be part of a "word".

The upper-case versions of these metasequences stand for the inversions of the corresponding character classes: `\D` matches a non-digit, `\S` a non-whitespace character, and `\W` a non-"word" character.

Following regexp custom, we identify "word" characters as `[A-Za-z0-9_]`, although these are too restrictive for what a Racketeer might consider a "word."

Remember to include a double backslash when putting these metasequences in a Racket string:

```
> (regexp-match #px"\\d\\d"
    "0 dear, 1 have 2 read catch 22 before 9")
'("22")
```

These character classes can be used inside a bracketed expression. For example, `#px"[a-z\\d]"` matches a lower-case letter or a digit.

9.4.2 POSIX character classes

A *POSIX character class* is a special metasequence of the form `[:...:]` that can be used only inside a bracketed expression in `#px` syntax. The POSIX classes supported are

- `[:alnum:]` — ASCII letters and digits
- `[:alpha:]` — ASCII letters
- `[:ascii:]` — ASCII characters
- `[:blank:]` — ASCII widthful whitespace: space and tab
- `[:cntrl:]` — "control" characters: ASCII 0 to 32
- `[:digit:]` — ASCII digits, same as `\d`
- `[:graph:]` — ASCII characters that use ink
- `[:lower:]` — ASCII lower-case letters
- `[:print:]` — ASCII ink-users plus widthful whitespace
- `[:space:]` — ASCII whitespace, same as `\s`
- `[:upper:]` — ASCII upper-case letters

- `[:word:]` — ASCII same as `\w`

- `[:xdigit:]` — ASCII hex digits

For example, the `#px"[[:alpha:]_]"` matches a letter or underscore.

```
> (regexp-match #px"[[:alpha:]_]" "--x--")
'("x")
> (regexp-match #px"[[:alpha:]_]" "--_--")
'("_")
> (regexp-match #px"[[:alpha:]_]" "--:--")
#f
```

The POSIX class notation is valid *only* inside a bracketed expression. For instance, `[:alpha:]`, when not inside a bracketed expression, will not be read as the letter class. Rather, it is (from previous principles) the character class containing the characters `:`, `a`, `l`, `p`, `h`.

```
> (regexp-match #px"[:alpha:]" "--a--")
'("a")
> (regexp-match #px"[:alpha:]" "--x--")
#f
```

9.5 Quantifiers

The *quantifiers* `*`, `+`, and `?` match respectively: zero or more, one or more, and zero or one instances of the preceding subpattern.

```
> (regexp-match-positions #rx"c[ad]*r" "cadaddadddr")
'((0 . 11))
> (regexp-match-positions #rx"c[ad]*r" "cr")
'((0 . 2))
> (regexp-match-positions #rx"c[ad]+r" "cadaddadddr")
'((0 . 11))
> (regexp-match-positions #rx"c[ad]+r" "cr")
#f
> (regexp-match-positions #rx"c[ad]?r" "cadaddadddr")
#f
> (regexp-match-positions #rx"c[ad]?r" "cr")
'((0 . 2))
> (regexp-match-positions #rx"c[ad]?r" "car")
'((0 . 3))
```

In `#px` syntax, you can use braces to specify much finer-tuned quantification than is possible with `*`, `+`, `?`:

- The quantifier `{`*m*`}` matches *exactly m* instances of the preceding subpattern; *m* must be a nonnegative integer.

- The quantifier `{`*m*`,`*n*`}` matches at least *m* and at most *n* instances. `m` and `n` are non-negative integers with *m* less or equal to *n*. You may omit either or both numbers, in which case *m* defaults to 0 and *n* to infinity.

It is evident that `+` and `?` are abbreviations for `{1,}` and `{0,1}` respectively, and `*` abbreviates `{,}`, which is the same as `{0,}`.

```
> (regexp-match #px"[aeiou]{3}" "vacuous")
'("uou")
> (regexp-match #px"[aeiou]{3}" "evolve")
#f
> (regexp-match #px"[aeiou]{2,3}" "evolve")
#f
> (regexp-match #px"[aeiou]{2,3}" "zeugma")
'("eu")
```

The quantifiers described so far are all *greedy*: they match the maximal number of instances that would still lead to an overall match for the full pattern.

```
> (regexp-match #rx"<.*>" "<tag1> <tag2> <tag3>")
'("<tag1> <tag2> <tag3>")
```

To make these quantifiers *non-greedy*, append a `?` to them. Non-greedy quantifiers match the minimal number of instances needed to ensure an overall match.

```
> (regexp-match #rx"<.*?>" "<tag1> <tag2> <tag3>")
'("<tag1>")
```

The non-greedy quantifiers are respectively: `*?`, `+?`, `??`, `{`*m*`}?`, `{`*m*`,`*n*`}?`. Note the two uses of the metacharacter `?`.

9.6 Clusters

Clustering—enclosure within parens `(...)`—identifies the enclosed *subpattern* as a single entity. It causes the matcher to capture the *submatch*, or the portion of the string matching the subpattern, in addition to the overall match:

```
> (regexp-match #rx"([a-z]+) ([0-9]+), ([0-9]+)" "jan 1, 1970")
'("jan 1, 1970" "jan" "1" "1970")
```

Clustering also causes a following quantifier to treat the entire enclosed subpattern as an entity:

```
> (regexp-match #rx"(poo )*" "poo poo platter")
'("poo poo " "poo ")
```

The number of submatches returned is always equal to the number of subpatterns specified in the regexp, even if a particular subpattern happens to match more than one substring or no substring at all.

```
> (regexp-match #rx"([a-z ]+;)*" "lather; rinse; repeat;")
'("lather; rinse; repeat;" " repeat;")
```

Here, the `*`-quantified subpattern matches three times, but it is the last submatch that is returned.

It is also possible for a quantified subpattern to fail to match, even if the overall pattern matches. In such cases, the failing submatch is represented by `#f`

```
> (define date-re
    ; match 'month year' or 'month day, year';
    ; subpattern matches day, if present
    #rx"([a-z]+) +([0-9]+,)? *([0-9]+)")
> (regexp-match date-re "jan 1, 1970")
'("jan 1, 1970" "jan" "1," "1970")
> (regexp-match date-re "jan 1970")
'("jan 1970" "jan" #f "1970")
```

9.6.1 Backreferences

Submatches can be used in the insert string argument of the procedures `regexp-replace` and `regexp-replace*`. The insert string can use \n as a *backreference* to refer back to the *n*th submatch, which is the substring that matched the *n*th subpattern. A \0 refers to the entire match, and it can also be specified as \&.

```
> (regexp-replace #rx"_(.+?)_"
    "the _nina_, the _pinta_, and the _santa maria_"
    "*\\1*")
"the *nina*, the _pinta_, and the _santa maria_"
> (regexp-replace* #rx"_(.+?)_"
    "the _nina_, the _pinta_, and the _santa maria_"
    "*\\1*")
"the *nina*, the *pinta*, and the *santa maria*"
> (regexp-replace #px"(\\S+) (\\S+) (\\S+)"
    "eat to live"
    "\\3 \\2 \\1")
"live to eat"
```

Use `\\` in the insert string to specify a literal backslash. Also, `\$` stands for an empty string, and is useful for separating a backreference `\n` from an immediately following number.

Backreferences can also be used within a `#px` pattern to refer back to an already matched subpattern in the pattern. `\n` stands for an exact repeat of the *n*th submatch. Note that `\0`, which is useful in an insert string, makes no sense within the regexp pattern, because the entire regexp has not matched yet that you could refer back to it.}

```
> (regexp-match #px"([a-z]+) and \\1"
                "billions and billions")
'("billions and billions" "billions")
```

Note that the backreference is not simply a repeat of the previous subpattern. Rather it is a repeat of the particular substring already matched by the subpattern.

In the above example, the backreference can only match `billions`. It will not match `millions`, even though the subpattern it harks back to—`([a-z]+)`—would have had no problem doing so:

```
> (regexp-match #px"([a-z]+) and \\1"
                "billions and millions")
#f
```

The following example corrects doubled words:

```
> (regexp-replace* #px"(\\S+) \\1"
     (string-append "now is the the time for all good men to "
                    "to come to the aid of of the party")
     "\\1")
"now is the time for all good men to come to the aid of the party"
```

The following example marks all immediately repeating patterns in a number string:

```
> (regexp-replace* #px"(\\d+)\\1"
     "12334098324243242098098234"
     "{\\1,\\1}")
"12{3,3}40983{24,24}3242{098,098}0234"
```

9.6.2 Non-capturing Clusters

It is often required to specify a cluster (typically for quantification) but without triggering the capture of submatch information. Such clusters are called *non-capturing*. To create a non-capturing cluster, use `(?:` instead of `(` as the cluster opener.

In the following example, a non-capturing cluster eliminates the "directory" portion of a given Unix pathname, and a capturing cluster identifies the basename.

But don't parse paths with regexps. Use functions like `split-path`, instead.

```
> (regexp-match #rx"^(?:[a-z]*/)*([a-z]+)$"
                "/usr/local/bin/racket")
'("/usr/local/bin/racket" "racket")
```

9.6.3 Cloisters

The location between the `?` and the `:` of a non-capturing cluster is called a *cloister*. You can put modifiers there that will cause the enclustered subpattern to be treated specially. The modifier `i` causes the subpattern to match case-insensitively:

The term *cloister* is a useful, if terminally cute, coinage from the abbots of Perl.

```
> (regexp-match #rx"(?i:hearth)" "HeartH")
'("HeartH")
```

The modifier `m` causes the subpattern to match in *multi-line mode*, where `.` does not match a newline character, `^` can match just after a newline, and `$` can match just before a newline.

```
> (regexp-match #rx"." "\na\n")
'("\n")
> (regexp-match #rx"(?m:.)" "\na\n")
'("a")
> (regexp-match #rx"^A plan$" "A man\nA plan\nA canal")
#f
> (regexp-match #rx"(?m:^A plan$)" "A man\nA plan\nA canal")
'("A plan")
```

You can put more than one modifier in the cloister:

```
> (regexp-match #rx"(?mi:^A Plan$)" "a man\na plan\na canal")
'("a plan")
```

A minus sign before a modifier inverts its meaning. Thus, you can use `-i` in a *subcluster* to overturn the case-insensitivities caused by an enclosing cluster.

```
> (regexp-match #rx"(?i:the (?-i:TeX)book)"
                "The TeXbook")
'("The TeXbook")
```

The above regexp will allow any casing for `the` and `book`, but it insists that `TeX` not be differently cased.

9.7 Alternation

You can specify a list of *alternate* subpatterns by separating them by `|`. The `|` separates subpatterns in the nearest enclosing cluster (or in the entire pattern string if there are no

179

enclosing parens).

```
> (regexp-match #rx"f(ee|i|o|um)" "a small, final fee")
'("fi" "i")
> (regexp-replace* #rx"([yi])s(e[sdr]?|ing|ation)"
                     (string-append
                      "analyse an energising organisation"
                      " pulsing with noisy organisms")
                     "\\1z\\2")
"analyze an energizing organization pulsing with noisy organisms"
```

Note again that if you wish to use clustering merely to specify a list of alternate subpatterns but do not want the submatch, use `(?:` instead of `(`.

```
> (regexp-match #rx"f(?:ee|i|o|um)" "fun for all")
'("fo")
```

An important thing to note about alternation is that the leftmost matching alternate is picked regardless of its length. Thus, if one of the alternates is a prefix of a later alternate, the latter may not have a chance to match.

```
> (regexp-match #rx"call|call-with-current-continuation"
                 "call-with-current-continuation")
'("call")
```

To allow the longer alternate to have a shot at matching, place it before the shorter one:

```
> (regexp-match #rx"call-with-current-continuation|call"
                 "call-with-current-continuation")
'("call-with-current-continuation")
```

In any case, an overall match for the entire regexp is always preferred to an overall non-match. In the following, the longer alternate still wins, because its preferred shorter prefix fails to yield an overall match.

```
> (regexp-match
   #rx"(?:call|call-with-current-continuation) constrained"
   "call-with-current-continuation constrained")
'("call-with-current-continuation constrained")
```

9.8   Backtracking

We've already seen that greedy quantifiers match the maximal number of times, but the overriding priority is that the overall match succeed. Consider

```
> (regexp-match #rx"a*a" "aaaa")
```

```
'("aaaa")
```

The regexp consists of two subregexps: `a*` followed by `a`. The subregexp `a*` cannot be allowed to match all four `a`'s in the text string `aaaa`, even though `*` is a greedy quantifier. It may match only the first three, leaving the last one for the second subregexp. This ensures that the full regexp matches successfully.

The regexp matcher accomplishes this via a process called *backtracking*. The matcher tentatively allows the greedy quantifier to match all four `a`'s, but then when it becomes clear that the overall match is in jeopardy, it *backtracks* to a less greedy match of three `a`'s. If even this fails, as in the call

```
> (regexp-match #rx"a*aa" "aaaa")
'("aaaa")
```

the matcher backtracks even further. Overall failure is conceded only when all possible backtracking has been tried with no success.

Backtracking is not restricted to greedy quantifiers. Nongreedy quantifiers match as few instances as possible, and progressively backtrack to more and more instances in order to attain an overall match. There is backtracking in alternation too, as the more rightward alternates are tried when locally successful leftward ones fail to yield an overall match.

Sometimes it is efficient to disable backtracking. For example, we may wish to commit to a choice, or we know that trying alternatives is fruitless. A nonbacktracking regexp is enclosed in `(?>...)`.

```
> (regexp-match #rx"(?>a+)." "aaaa")
#f
```

In this call, the subregexp `?>a+` greedily matches all four `a`'s, and is denied the opportunity to backtrack. So, the overall match is denied. The effect of the regexp is therefore to match one or more `a`'s followed by something that is definitely non-`a`.

9.9   Looking Ahead and Behind

You can have assertions in your pattern that look *ahead* or *behind* to ensure that a subpattern does or does not occur. These "look around" assertions are specified by putting the subpattern checked for in a cluster whose leading characters are: `?=` (for positive lookahead), `?!` (negative lookahead), `?<=` (positive lookbehind), `?<!` (negative lookbehind). Note that the subpattern in the assertion does not generate a match in the final result; it merely allows or disallows the rest of the match.

9.9.1 Lookahead

Positive lookahead with `?=` peeks ahead to ensure that its subpattern *could* match.

```
> (regexp-match-positions #rx"grey(?=hound)"
    "i left my grey socks at the greyhound")
'((28 . 32))
```

The regexp `#rx"grey(?=hound)"` matches `grey`, but *only* if it is followed by `hound`. Thus, the first `grey` in the text string is not matched.

Negative lookahead with `?!` peeks ahead to ensure that its subpattern *could not* possibly match.

```
> (regexp-match-positions #rx"grey(?!hound)"
    "the gray greyhound ate the grey socks")
'((27 . 31))
```

The regexp `#rx"grey(?!hound)"` matches `grey`, but only if it is *not* followed by `hound`. Thus the `grey` just before `socks` is matched.

9.9.2 Lookbehind

Positive lookbehind with `?<=` checks that its subpattern *could* match immediately to the left of the current position in the text string.

```
> (regexp-match-positions #rx"(?<=grey)hound"
    "the hound in the picture is not a greyhound")
'((38 . 43))
```

The regexp `#rx"(?<=grey)hound"` matches `hound`, but only if it is preceded by `grey`.

Negative lookbehind with `?<!` checks that its subpattern could not possibly match immediately to the left.

```
> (regexp-match-positions #rx"(?<!grey)hound"
    "the greyhound in the picture is not a hound")
'((38 . 43))
```

The regexp `#rx"(?<!grey)hound"` matches `hound`, but only if it is *not* preceded by `grey`.

Lookaheads and lookbehinds can be convenient when they are not confusing.

9.10 An Extended Example

Here's an extended example from Friedl's *Mastering Regular Expressions*, page 189, that covers many of the features described in this chapter. The problem is to fashion a regexp that will match any and only IP addresses or *dotted quads*: four numbers separated by three dots, with each number between 0 and 255.

First, we define a subregexp `n0-255` that matches 0 through 255:

```
> (define n0-255
    (string-append
     "(?:"
     "\\d|"          ;  0 through 9
     "\\d\\d|"       ;  00 through 99
     "[01]\\d\\d|"   ; 000 through 199
     "2[0-4]\\d|"    ; 200 through 249
     "25[0-5]"       ; 250 through 255
     ")"))
```

The first two alternates simply get all single- and double-digit numbers. Since 0-padding is allowed, we need to match both 1 and 01. We need to be careful when getting 3-digit numbers, since numbers above 255 must be excluded. So we fashion alternates to get 000 through 199, then 200 through 249, and finally 250 through 255.

Note that `n0-255` lists prefixes as preferred alternates, which is something we cautioned against in §9.7 "Alternation". However, since we intend to anchor this subregexp explicitly to force an overall match, the order of the alternates does not matter.

An IP-address is a string that consists of four `n0-255`s with three dots separating them.

```
> (define ip-re1
    (string-append
     "^"           ; nothing before
     n0-255        ; the first n0-255,
     "(?:"         ; then the subpattern of
     "\\."         ; a dot followed by
     n0-255        ; an n0-255,
     ")"           ; which is
     "{3}"         ; repeated exactly 3 times
     "$"))         ; with nothing following))
```

Let's try it out:

```
> (regexp-match (pregexp ip-re1) "1.2.3.4")
'("1.2.3.4")
> (regexp-match (pregexp ip-re1) "55.155.255.265")
#f
```

which is fine, except that we also have

```
> (regexp-match (pregexp ip-re1) "0.00.000.00")
```

```
'("0.00.000.00")
```

All-zero sequences are not valid IP addresses! Lookahead to the rescue. Before starting to match `ip-re1`, we look ahead to ensure we don't have all zeros. We could use positive lookahead to ensure there *is* a digit other than zero.

```
> (define ip-re
    (pregexp
     (string-append
       "(?=.*[1-9])" ; ensure there's a non-0 digit
       ip-re1)))
```

Or we could use negative lookahead to ensure that what's ahead isn't composed of *only* zeros and dots.

```
> (define ip-re
    (pregexp
     (string-append
       "(?![0.]*$)" ; not just zeros and dots
                    ; (note: . is not metachar inside [...])
       ip-re1)))
```

The regexp `ip-re` will match all and only valid IP addresses.

```
> (regexp-match ip-re "1.2.3.4")
'("1.2.3.4")
> (regexp-match ip-re "0.0.0.0")
#f
```

10   Exceptions and Control

Racket provides an especially rich set of control operations—not only operations for raising and catching exceptions, but also operations for grabbing and restoring portions of a computation.

10.1   Exceptions

Whenever a run-time error occurs, an *exception* is raised. Unless the exception is caught, then it is handled by printing a message associated with the exception, and then escaping from the computation.

```
> (/ 1 0)
/: division by zero
> (car 17)
car: expects argument of type <pair>; given 17
```

To catch an exception, use the `with-handlers` form:

```
(with-handlers ([predicate-expr handler-expr] ...)
  body ...+)
```

Each `predicate-expr` in a handler determines a kind of exception that is caught by the `with-handlers` form, and the value representing the exception is passed to the handler procedure produced by `handler-expr`. The result of the `handler-expr` is the result of the `with-handlers` expression.

For example, a divide-by-zero error raises an instance of the `exn:fail:contract:divide-by-zero` structure type:

```
> (with-handlers ([exn:fail:contract:divide-by-zero?
                   (lambda (exn) +inf.0)])
    (/ 1 0))
+inf.0
> (with-handlers ([exn:fail:contract:divide-by-zero?
                   (lambda (exn) +inf.0)])
    (car 17))
car: expects argument of type <pair>; given 17
```

The `error` function is one way to raise your own exception. It packages an error message and other information into an `exn:fail` structure:

```
> (error "crash!")
```

```
crash!
> (with-handlers ([exn:fail? (lambda (exn) 'air-bag)])
    (error "crash!"))
'air-bag
```

The `exn:fail:contract:divide-by-zero` and `exn:fail` structure types are sub-types of the `exn` structure type. Exceptions raised by core forms and functions always raise an instance of `exn` or one of its sub-types, but an exception does not have to be represented by a structure. The `raise` function lets you raise any value as an exception:

```
> (raise 2)
uncaught exception: 2
> (with-handlers ([(lambda (v) (equal? v 2)) (lambda (v) 'two)])
    (raise 2))
'two
> (with-handlers ([(lambda (v) (equal? v 2)) (lambda (v) 'two)])
    (/ 1 0))
/: division by zero
```

Multiple *predicate-expr*s in a `with-handlers` form let you handle different kinds of exceptions in different ways. The predicates are tried in order, and if none of them match, then the exception is propagated to enclosing contexts.

```
> (define (always-fail n)
    (with-handlers ([even? (lambda (v) 'even)]
                    [positive? (lambda (v) 'positive)])
      (raise n)))
> (always-fail 2)
'even
> (always-fail 3)
'positive
> (always-fail -3)
uncaught exception: -3
> (with-handlers ([negative? (lambda (v) 'negative)])
    (always-fail -3))
'negative
```

Using `(lambda (v) #t)` as a predicate captures all exceptions, of course:

```
> (with-handlers ([(lambda (v) #t) (lambda (v) 'oops)])
    (car 17))
'oops
```

Capturing all exceptions is usually a bad idea, however. If the user types Ctl-C in a terminal window or clicks the Stop button in DrRacket to interrupt a computation, then normally the `exn:break` exception should not be caught. To catch only exceptions that represent errors, use `exn:fail?` as the predicate:

```
> (with-handlers ([exn:fail? (lambda (v) 'oops)])
    (car 17))
'oops
> (with-handlers ([exn:fail? (lambda (v) 'oops)])
    (break-thread (current-thread)) ; simulate Ctl-C
    (car 17))
user break
```

10.2 Prompts and Aborts

When an exception is raised, control escapes out of an arbitrary deep evaluation context to
the point where the exception is caught—or all the way out if the expression is never caught:

```
> (+ 1 (+ 1 (+ 1 (+ 1 (+ 1 (+ 1 (/ 1 0)))))))
/: division by zero
```

But if control escapes "all the way out," why does the REPL keep going after an error is
printed? You might think that it's because the REPL wraps every interaction in a `with-handlers` form that catches all exceptions, but that's not quite the reason.

The actual reason is that the REPL wraps the interaction with a *prompt*, which effectively
marks the evaluation context with an escape point. If an exception is not caught, then in-
formation about the exception is printed, and then evaluation *aborts* to the nearest enclosing
prompt. More precisely, each prompt has a *prompt tag*, and there is a designated *default
prompt tag* that the uncaught-exception handler uses to abort.

The `call-with-continuation-prompt` function installs a prompt with a given prompt
tag, and then it evaluates a given thunk under the prompt. The `default-continuation-
prompt-tag` function returns the default prompt tag. The `abort-current-continuation`
function escapes to the nearest enclosing prompt that has a given prompt tag.

```
> (define (escape v)
    (abort-current-continuation
     (default-continuation-prompt-tag)
     (lambda () v)))
> (+ 1 (+ 1 (+ 1 (+ 1 (+ 1 (+ 1 (escape 0)))))))
0
> (+ 1
    (call-with-continuation-prompt
     (lambda ()
       (+ 1 (+ 1 (+ 1 (+ 1 (+ 1 (+ 1 (escape 0))))))))
     (default-continuation-prompt-tag)))
1
```

187

In `escape` above, the value `v` is wrapped in a procedure that is called after escaping to the enclosing prompt.

Prompts and aborts look very much like exception handling and raising. Indeed, prompts and aborts are essentially a more primitive form of exceptions, and `with-handlers` and `raise` are implemented in terms of prompts and aborts. The power of the more primitive forms is related to the word "continuation" in the operator names, as we discuss in the next section.

10.3 Continuations

A *continuation* is a value that encapsulates a piece of an expression context. The `call-with-composable-continuation` function captures the *current continuation* starting outside the current function call and running up to the nearest enclosing prompt. (Keep in mind that each REPL interaction is implicitly wrapped in a prompt.)

For example, in

```
(+ 1 (+ 1 (+ 1 0)))
```

at the point where `0` is evaluated, the expression context includes three nested addition expressions. We can grab that context by changing `0` to grab the continuation before returning `0`:

```
> (define saved-k #f)
> (define (save-it!)
    (call-with-composable-continuation
     (lambda (k) ; k is the captured continuation
       (set! saved-k k)
       0)))
> (+ 1 (+ 1 (+ 1 (save-it!))))
3
```

The continuation saved in `save-k` encapsulates the program context `(+ 1 (+ 1 (+ 1 ?)))`, where `?` represents a place to plug in a result value—because that was the expression context when `save-it!` was called. The continuation is encapsulated so that it behaves like the function `(lambda (v) (+ 1 (+ 1 (+ 1 v))))`:

```
> (saved-k 0)
3
> (saved-k 10)
13
> (saved-k (saved-k 0))
6
```

The continuation captured by `call-with-composable-continuation` is determined dynamically, not syntactically. For example, with

```
> (define (sum n)
    (if (zero? n)
        (save-it!)
        (+ n (sum (sub1 n)))))
> (sum 5)
15
```

the continuation in `saved-k` becomes `(lambda (x) (+ 5 (+ 4 (+ 3 (+ 2 (+ 1 x))))))`:

```
> (saved-k 0)
15
> (saved-k 10)
25
```

A more traditional continuation operator in Racket is `call-with-current-continuation`, which is often abbreviated `call/cc`. It is like `call-with-composable-continuation`, but applying the captured continuation first aborts (to the current prompt) before restoring the saved continuation. In addition, Racket systems traditionally support a single prompt at the program start, instead of allowing new prompts via `call-with-continuation-prompt`. Continuations as in Racket are sometimes called *delimited continuations*, since a program can introduce new delimiting prompts, and continuations as captured by `call-with-composable-continuation` are sometimes called *composable continuations*, because they do not have a built-in abort.

For an example of how continuations are useful, see ***More***: *Systems Programming with Racket*. For specific control operators that have more convenient names than the primitives described here, see `racket/control`.

11 Iterations and Comprehensions

The `for` family of syntactic forms support iteration over *sequences*. Lists, vectors, strings, byte strings, input ports, and hash tables can all be used as sequences, and constructors like `in-range` offer even more kinds of sequences.

Variants of `for` accumulate iteration results in different ways, but they all have the same syntactic shape. Simplifying for now, the syntax of `for` is

```
(for ([id sequence-expr] ...)
  body ...+)
```

A `for` loop iterates through the sequence produced by the `sequence-expr`. For each element of the sequence, `for` binds the element to `id`, and then it evaluates the `body`s for side effects.

Examples:
```
> (for ([i '(1 2 3)])
    (display i))
123
> (for ([i "abc"])
    (printf "~a..." i))
a...b...c...
```

The `for/list` variant of `for` is more Racket-like. It accumulates `body` results into a list, instead of evaluating `body` only for side effects. In more technical terms, `for/list` implements a *list comprehension*.

Examples:
```
> (for/list ([i '(1 2 3)])
    (* i i))
'(1 4 9)
> (for/list ([i "abc"])
    i)
'(#\a #\b #\c)
```

The full syntax of `for` accommodates multiple sequences to iterate in parallel, and the `for*` variant nests the iterations instead of running them in parallel. More variants of `for` and `for*` accumulate `body` results in different ways. In all of these variants, predicates that prune iterations can be included along with bindings.

Before details on the variations of `for`, though, it's best to see the kinds of sequence generators that make interesting examples.

11.1 Sequence Constructors

The `in-range` function generates a sequence of numbers, given an optional starting number (which defaults to 0), a number before which the sequences ends, and an optional step (which defaults to 1).

Examples:
```
> (for ([i (in-range 3)])
    (display i))
012
> (for ([i (in-range 1 4)])
    (display i))
123
> (for ([i (in-range 1 4 2)])
    (display i))
13
> (for ([i (in-range 4 1 -1)])
    (display i))
432
> (for ([i (in-range 1 4 1/2)])
    (printf " ~a " i))
 1  3/2  2  5/2  3  7/2
```

The `in-naturals` function is similar, except that the starting number must be an exact non-negative integer (which defaults to 0), the step is always 1, and there is no upper limit. A `for` loop using just `in-naturals` will never terminate unless a body expression raises an exception or otherwise escapes.

Example:
```
> (for ([i (in-naturals)])
    (if (= i 10)
        (error "too much!")
        (display i)))
0123456789
too much!
```

The `stop-before` and `stop-after` functions construct a new sequence given a sequence and a predicate. The new sequence is like the given sequence, but truncated either immediately before or immediately after the first element for which the predicate returns true.

Example:
```
> (for ([i (stop-before "abc def"
                        char-whitespace?)])
    (display i))
abc
```

Sequence constructors like `in-list`, `in-vector` and `in-string` simply make explicit the use of a list, vector, or string as a sequence. Since they raise an exception when given the wrong kind of value, and since they otherwise avoid a run-time dispatch to determine the sequence type, they enable more efficient code generation; see §11.9 "Iteration Performance" for more information.

Examples:

```
> (for ([i (in-string "abc")])
    (display i))
abc
> (for ([i (in-string '(1 2 3))])
    (display i))
raise-type-error: expects type <symbol> as 1st argument,
given: #<procedure:in-string>; other arguments were:
"string" '(1 2 3)
```

11.2 `for` and `for*`

A more complete syntax of `for` is

```
(for (clause ...)
  body ...+)

clause = [id sequence-expr]
       | #:when boolean-expr
```

When multiple `[id sequence-expr]` clauses are provided in a `for` form, the corresponding sequences are traversed in parallel:

```
> (for ([i (in-range 1 4)]
        [chapter '("Intro" "Details" "Conclusion")])
    (printf "Chapter ~a. ~a\n" i chapter))
Chapter 1. Intro
Chapter 2. Details
Chapter 3. Conclusion
```

With parallel sequences, the `for` expression stops iterating when any sequence ends. This behavior allows `in-naturals`, which creates an infinite sequence of numbers, to be used for indexing:

```
> (for ([i (in-naturals 1)]
        [chapter '("Intro" "Details" "Conclusion")])
    (printf "Chapter ~a. ~a\n" i chapter))
```

192

```
Chapter 1. Intro
Chapter 2. Details
Chapter 3. Conclusion
```

The `for*` form, which has the same syntax as `for`, nests multiple sequences instead of running them in parallel:

```
> (for* ([book '("Guide" "Reference")]
         [chapter '("Intro" "Details" "Conclusion")])
    (printf "~a ~a\n" book chapter))
Guide Intro
Guide Details
Guide Conclusion
Reference Intro
Reference Details
Reference Conclusion
```

Thus, `for*` is a shorthand for nested `for`s in the same way that `let*` is a shorthand for nested `let`s.

The `#:when` *boolean-expr* form of a *clause* is another shorthand. It allows the *body*s to evaluate only when the *boolean-expr* produces a true value:

```
> (for* ([book '("Guide" "Reference")]
         [chapter '("Intro" "Details" "Conclusion")]
         #:when (not (equal? chapter "Details")))
    (printf "~a ~a\n" book chapter))
Guide Intro
Guide Conclusion
Reference Intro
Reference Conclusion
```

A *boolean-expr* with `#:when` can refer to any of the preceding iteration bindings. In a `for` form, this scoping makes sense only if the test is nested in the iteration of the preceding bindings; thus, bindings separated by `#:when` are mutually nested, instead of in parallel, even with `for`.

```
> (for ([book '("Guide" "Reference" "Notes")]
        #:when (not (equal? book "Notes"))
        [i (in-naturals 1)]
        [chapter '("Intro" "Details" "Conclusion" "Index")]
        #:when (not (equal? chapter "Index")))
    (printf "~a Chapter ~a. ~a\n" book i chapter))
Guide Chapter 1. Intro
Guide Chapter 2. Details
Guide Chapter 3. Conclusion
Reference Chapter 1. Intro
```

11.3 `for/list` and `for*/list`

The `for/list` form, which has the same syntax as `for`, evaluates the *body*s to obtain values that go into a newly constructed list:

```
> (for/list ([i (in-naturals 1)]
             [chapter '("Intro" "Details" "Conclusion")])
    (string-append (number->string i) ". " chapter))
'("1. Intro" "2. Details" "3. Conclusion")
```

A `#:when` clause in a `for-list` form prunes the result list along with evaluations of the *body*s:

```
> (for/list ([i (in-naturals 1)]
             [chapter '("Intro" "Details" "Conclusion")]
             #:when (odd? i))
    chapter)
'("Intro" "Conclusion")
```

This pruning behavior of `#:when` is more useful with `for/list` than `for`. Whereas a plain `when` form normally suffices with `for`, a `when` expression form in a `for/list` would cause the result list to contain `#<void>`s instead of omitting list elements.

The `for*/list` form is like `for*`, nesting multiple iterations:

```
> (for*/list ([book '("Guide" "Ref.")]
              [chapter '("Intro" "Details")])
    (string-append book " " chapter))
'("Guide Intro" "Guide Details" "Ref. Intro" "Ref. Details")
```

A `for*/list` form is not quite the same thing as nested `for/list` forms. Nested `for/list`s would produce a list of lists, instead of one flattened list. Much like `#:when`, then, the nesting of `for*/list` is more useful than the nesting of `for*`.

11.4 `for/vector` and `for*/vector`

The `for/vector` form can be used with the same syntax as the `for/list` form, but the evaluated *body*s go into a newly-constructed vector instead of a list:

```
> (for/vector ([i (in-naturals 1)]
               [chapter '("Intro" "Details" "Conclusion")])
```

194

```
      (string-append (number->string i) ". " chapter))
  '#("1. Intro" "2. Details" "3. Conclusion")
```

The `for*/vector` form behaves similarly, but the iterations are nested as in `for*`.

The `for/vector` and `for*/vector` forms also allow the length of the vector to be constructed to be supplied in advance. The resulting iteration can be performed more efficiently than plain `for/vector` or `for*/vector`:

```
  > (let ((chapters '("Intro" "Details" "Conclusion")))
      (for/vector #:length (length chapters) ([i (in-naturals 1)]
                                              [chapter chapters])
        (string-append (number->string i) ". " chapter)))
  '#("1. Intro" "2. Details" "3. Conclusion")
```

If a length is provided, the iteration stops when the vector is filled or the requested iterations are complete, whichever comes first. If the provided length exceeds the requested number of iterations, then the remaining slots in the vector are initialized to the default argument of `make-vector`.

11.5 `for/and` **and** `for/or`

The `for/and` form combines iteration results with `and`, stopping as soon as it encounters `#f`:

```
  > (for/and ([chapter '("Intro" "Details" "Conclusion")])
      (equal? chapter "Intro"))
  #f
```

The `for/or` form combines iteration results with `or`, stopping as soon as it encounters a true value:

```
  > (for/or ([chapter '("Intro" "Details" "Conclusion")])
      (equal? chapter "Intro"))
  #t
```

As usual, the `for*/and` and `for*/or` forms provide the same facility with nested iterations.

11.6 `for/first` **and** `for/last`

The `for/first` form returns the result of the first time that the *body*s are evaluated, skipping further iterations. This form is most useful with a `#:when` clause.

```
  > (for/first ([chapter '("Intro" "Details" "Conclusion" "Index")]
```

```
                      #:when (not (equal? chapter "Intro")))
        chapter)
  "Details"
```

If the *body*s are evaluated zero times, then the result is #f.

The for/last form runs all iterations, returning the value of the last iteration (or #f if no iterations are run):

```
  > (for/last ([chapter '("Intro" "Details" "Conclusion" "Index")]
                      #:when (not (equal? chapter "Index")))
        chapter)
  "Conclusion"
```

As usual, the for*/first and for*/last forms provide the same facility with nested iterations:

```
  > (for*/first ([book '("Guide" "Reference")]
                     [chapter '("Intro" "Details" "Conclusion" "Index")]
                     #:when (not (equal? chapter "Intro")))
        (list book chapter))
  '("Guide" "Details")
  > (for*/last ([book '("Guide" "Reference")]
                     [chapter '("Intro" "Details" "Conclusion" "Index")]
                     #:when (not (equal? chapter "Index")))
        (list book chapter))
  '("Reference" "Conclusion")
```

11.7   for/fold **and** for*/fold

The for/fold form is a very general way to combine iteration results. Its syntax is slightly different than the syntax of for, because accumulation variables must be declared at the beginning:

```
  (for/fold ([accum-id init-expr] ...)
                (clause ...)
      body ...+)
```

In the simple case, only one [accum-id init-expr] is provided, and the result of the for/fold is the final value for *accum-id*, which starts out with the value of *init-expr*. In the *clause*s and *body*s, *accum-id* can be referenced to get its current value, and the last *body* provides the value of *accum-id* for the next iteration.

Examples:
```
  > (for/fold ([len 0])
```

```
            ([chapter '("Intro" "Conclusion")])
      (+ len (string-length chapter)))
  15
> (for/fold ([prev #f])
            ([i (in-naturals 1)]
             [chapter '("Intro" "Details" "Details" "Conclusion")]
             #:when (not (equal? chapter prev)))
      (printf "~a. ~a\n" i chapter)
      chapter)
1. Intro
2. Details
4. Conclusion
"Conclusion"
```

When multiple *accum-id*s are specified, then the last *body* must produce multiple values, one for each *accum-id*. The `for/fold` expression itself produces multiple values for the results.

Example:
```
> (for/fold ([prev #f]
             [counter 1])
            ([chapter '("Intro" "Details" "Details" "Conclusion")]
             #:when (not (equal? chapter prev)))
      (printf "~a. ~a\n" counter chapter)
      (values chapter
              (add1 counter)))
1. Intro
2. Details
3. Conclusion
"Conclusion"
4
```

11.8   Multiple-Valued Sequences

In the same way that a function or expression can produce multiple values, individual iterations of a sequence can produce multiple elements. For example, a hash table as a sequence generates two values for each iteration: a key and a value.

In the same way that `let-values` binds multiple results to multiple identifiers, `for` can bind multiple sequence elements to multiple iteration identifiers:

While `let` must be changed to `let-values` to bind multiple identifier, `for` simply allows a parenthesized list of identifiers instead of a single identifier in any clause.

```
> (for ([(k v) #hash(("apple" . 1) ("banana" . 3))])
      (printf "~a count: ~a\n" k v))
apple count: 1
banana count: 3
```

This extension to multiple-value bindings works for all `for` variants. For example, `for*/list` nests iterations, builds a list, and also works with multiple-valued sequences:

```
> (for*/list ([(k v) #hash(("apple" . 1) ("banana" . 3))]
              [(i) (in-range v)])
    k)
'("apple" "banana" "banana" "banana")
```

11.9   Iteration Performance

Ideally, a `for` iteration should run as fast as a loop that you write by hand as a recursive-function invocation. A hand-written loop, however, is normally specific to a particular kind of data, such as lists. In that case, the hand-written loop uses selectors like `car` and `cdr` directly, instead of handling all forms of sequences and dispatching to an appropriate iterator.

The `for` forms can provide the performance of hand-written loops when enough information is apparent about the sequences to iterate. Specifically, the clause should have one of the following *fast-clause* forms:

```
fast-clause = [id fast-seq]
            | [(id) fast-seq]
            | [(id id) fast-indexed-seq]
            | [(id ...) fast-parallel-seq]

fast-seq = (in-range expr expr)
         | (in-range expr expr expr)
         | (in-naturals)
         | (in-naturals expr)
         | (in-list expr)
         | (in-vector expr)
         | (in-string expr)
         | (in-bytes expr)
         | (in-value expr)
         | (stop-before fast-seq predicate-expr)
         | (stop-after fast-seq predicate-expr)

fast-indexed-seq = (in-indexed fast-seq)
                 | (stop-before fast-indexed-seq predicate-expr)
                 | (stop-after fast-indexed-seq predicate-expr)

fast-parallel-seq = (in-parallel fast-seq ...)
                  | (stop-before fast-parallel-seq predicate-expr)
                  | (stop-after fast-parallel-seq predicate-expr)
```

Examples:

```
> (time (for ([i (in-range 100000)])
          (for ([elem (in-list '(a b c d e f g h))]) ; fast
            (void))))
cpu time: 3 real time: 3 gc time: 0
> (time (for ([i (in-range 100000)])
          (for ([elem '(a b c d e f g h)])            ; slower
            (void))))
cpu time: 55 real time: 54 gc time: 0
> (time (let ([seq (in-list '(a b c d e f g h))])
          (for ([i (in-range 100000)])
            (for ([elem seq])                         ; slower
              (void)))))
cpu time: 59 real time: 59 gc time: 0
```

The grammars above are not complete, because the set of syntactic patterns that provide
good performance is extensible, just like the set of sequence values. The documentation for
a sequence constructor should indicate the performance benefits of using it directly in a for
clause.

§2.18 "Iterations
and
Comprehensions:
for, for/list, ..."
in *Reference:
Racket* provides
more on iterations
and
comprehensions.

199

12 Pattern Matching

The `match` form supports pattern matching on arbitrary Racket values, as opposed to functions like `regexp-match` that compare regular expressions to byte and character sequences (see §9 "Regular Expressions").

```
(match target-expr
  [pattern expr ...+] ...)
```

The `match` form takes the result of `target-expr` and tries to match each `pattern` in order. As soon as it finds a match, it evaluates the corresponding `expr` sequence to obtain the result for the `match` form. If `pattern` includes *pattern variables*, they are treated like wildcards, and each variable is bound in the `expr` to the input fragments that it matched.

Most Racket literal expressions can be used as patterns:

```
> (match 2
    [1 'one]
    [2 'two]
    [3 'three])
'two
> (match #f
    [#t 'yes]
    [#f 'no])
'no
> (match "apple"
    ['apple 'symbol]
    ["apple" 'string]
    [#f 'boolean])
'string
```

Constructors like `cons`, `list`, and `vector` can be used to create patterns that match pairs, lists, and vectors:

```
> (match '(1 2)
    [(list 0 1) 'one]
    [(list 1 2) 'two])
'two
> (match '(1 . 2)
    [(list 1 2) 'list]
    [(cons 1 2) 'pair])
'pair
> (match #(1 2)
    [(list 1 2) 'list]
```

```
    [(vector 1 2) 'vector])
'vector
```

A constructor bound with struct also can be used as a pattern constructor:

```
> (struct shoe (size color))
> (struct hat (size style))
> (match (hat 23 'bowler)
    [(shoe 10 'white) "bottom"]
    [(hat 23 'bowler) "top"])
"top"
```

Unquoted, non-constructor identifiers in a pattern are pattern variables that are bound in the result expressions:

```
> (match '(1)
    [(list x) (+ x 1)]
    [(list x y) (+ x y)])
2
> (match '(1 2)
    [(list x) (+ x 1)]
    [(list x y) (+ x y)])
3
> (match (hat 23 'bowler)
    [(shoe sz col) sz]
    [(hat sz stl) sz])
23
```

An ellipsis, written ..., act like a Kleene star within a list or vector pattern: the preceding sub-pattern can be used to match any number of times for any number of consecutive elements of the list of vector. If a sub-pattern followed by an ellipsis includes a pattern variable, the variable matches multiple times, and it is bound in the result expression to a list of matches:

```
> (match '(1 1 1)
    [(list 1 ...) 'ones]
    [else 'other])
'ones
> (match '(1 1 2)
    [(list 1 ...) 'ones]
    [else 'other])
'other
> (match '(1 2 3 4)
    [(list 1 x ... 4) x])
'(2 3)
> (match (list (hat 23 'bowler) (hat 22 'pork-pie))
    [(list (hat sz styl) ...) (apply + sz)])
```

Ellipses can be nested to match nested repetitions, and in that case, pattern variables can be bound to lists of lists of matches:

```
> (match '((! 1) (! 2 2) (! 3 3 3))
    [(list (list '! x ...) ...) x])
'((1) (2 2) (3 3 3))
```

The quasiquote form (see §4.11 "Quasiquoting: quasiquote and '" for more about it) can also be used to build patterns. While unquoted portions of a normal quasiquoted form mean regular racket evaluation, here unquoted portions mean go back to regular pattern matching.

So, in the example below, the with expression is the pattern and it gets rewritten into the application expression, using quasiquote as a pattern in the first instance and quasiquote to build an expression in the second.

```
> (match '{with {x 1} {+ x 1}}
    ['{with {,id ,rhs} ,body}
     '{{lambda {,id} ,body} ,rhs}])
'((lambda (x) (+ x 1)) 1)
```

For information on many more pattern forms, see racket/match.

Forms like match-let and match-lambda support patterns in positions that otherwise must be identifiers. For example, match-let generalizes let to a destructing bind:

```
> (match-let ([(list x y z) '(1 2 3)])
    (list z y x))
'(3 2 1)
```

For information on these additional forms, see racket/match.

§8 "Pattern Matching" in *Reference: Racket* provides more on pattern matching.

13 Classes and Objects

A `class` expression denotes a first-class value, just like a `lambda` expression:

```
(class superclass-expr decl-or-expr ...)
```

The *superclass-expr* determines the superclass for the new class. Each *decl-or-expr* is either a declaration related to methods, fields, and initialization arguments, or it is an expression that is evaluated each time that the class is instantiated. In other words, instead of a method-like constructor, a class has initialization expressions interleaved with field and method declarations.

By convention, class names end with `%`. The built-in root class is `object%`. The following expression creates a class with public methods `get-size`, `grow`, and `eat`:

```
(class object%
  (init size)                 ; initialization argument

  (define current-size size)  ; field

  (super-new)                 ; superclass initialization

  (define/public (get-size)
    current-size)

  (define/public (grow amt)
    (set! current-size (+ amt current-size)))

  (define/public (eat other-fish)
    (grow (send other-fish get-size))))
```

The `size` initialization argument must be supplied via a named argument when instantiating the class through the `new` form:

```
(new (class object% (init size) ....) [size 10])
```

Of course, we can also name the class and its instance:

```
(define fish% (class object% (init size) ....))
(define charlie (new fish% [size 10]))
```

In the definition of `fish%`, `current-size` is a private field that starts out with the value of the `size` initialization argument. Initialization arguments like `size` are available only during class instantiation, so they cannot be referenced directly from a method. The `current-size` field, in contrast, is available to methods.

203

The (super-new) expression in `fish%` invokes the initialization of the superclass. In this case, the superclass is `object%`, which takes no initialization arguments and performs no work; `super-new` must be used, anyway, because a class must always invoke its superclass's initialization.

Initialization arguments, field declarations, and expressions such as (super-new) can appear in any order within a `class`, and they can be interleaved with method declarations. The relative order of expressions in the class determines the order of evaluation during instantiation. For example, if a field's initial value requires calling a method that works only after superclass initialization, then the field declaration must be placed after the `super-new` call. Ordering field and initialization declarations in this way helps avoid imperative assignment. The relative order of method declarations makes no difference for evaluation, because methods are fully defined before a class is instantiated.

13.1 Methods

Each of the three `define/public` declarations in `fish%` introduces a new method. The declaration uses the same syntax as a Racket function, but a method is not accessible as an independent function. A call to the `grow` method of a `fish%` object requires the `send` form:

```
> (send charlie grow 6)
> (send charlie get-size)
16
```

Within `fish%`, self methods can be called like functions, because the method names are in scope. For example, the `eat` method within `fish%` directly invokes the `grow` method. Within a class, attempting to use a method name in any way other than a method call results in a syntax error.

In some cases, a class must call methods that are supplied by the superclass but not overridden. In that case, the class can use `send` with `this` to access the method:

```
(define hungry-fish% (class fish% (super-new)
                       (define/public (eat-more fish1 fish2)
                         (send this eat fish1)
                         (send this eat fish2))))
```

Alternately, the class can declare the existence of a method using `inherit`, which brings the method name into scope for a direct call:

```
(define hungry-fish% (class fish% (super-new)
                       (inherit eat)
                       (define/public (eat-more fish1 fish2)
```

```
                       (eat fish1) (eat fish2))))
```

With the `inherit` declaration, if `fish%` had not provided an `eat` method, an error would be signaled in the evaluation of the `class` form for `hungry-fish%`. In contrast, with `(send this ....)`, an error would not be signaled until the `eat-more` method is called and the `send` form is evaluated. For this reason, `inherit` is preferred.

Another drawback of `send` is that it is less efficient than `inherit`. Invocation of a method via `send` involves finding a method in the target object's class at run time, making `send` comparable to an interface-based method call in Java. In contrast, `inherit`-based method invocations use an offset within the class's method table that is computed when the class is created.

To achieve performance similar to `inherit`-based method calls when invoking a method from outside the method's class, the programmer must use the `generic` form, which produces a class- and method-specific *generic method* to be invoked with `send-generic`:

```
(define get-fish-size (generic fish% get-size))

> (send-generic charlie get-fish-size)
16
> (send-generic (new hungry-fish% [size 32]) get-fish-size)
32
> (send-generic (new object%) get-fish-size)
```
generic:get-size for class: fish%: expected argument of
type <instance for class: fish%>; given (object)

Roughly speaking, the form translates the class and the external method name to a location in the class's method table. As illustrated by the last example, sending through a generic method checks that its argument is an instance of the generic's class.

Whether a method is called directly within a `class`, through a generic method, or through `send`, method overriding works in the usual way:

```
(define picky-fish% (class fish% (super-new)
                       (define/override (grow amt)

                         (super grow (* 3/4 amt)))))
(define daisy (new picky-fish% [size 20]))

> (send daisy eat charlie)
> (send daisy get-size)
32
```

The grow method in `picky-fish%` is declared with `define/override` instead of `define/public`, because grow is meant as an overriding declaration. If grow had been declared with `define/public`, an error would have been signaled when evaluating the `class` expression, because `fish%` already supplies grow.

Using `define/override` also allows the invocation of the overridden method via a super call. For example, the grow implementation in `picky-fish%` uses super to delegate to the superclass implementation.

13.2   Initialization Arguments

Since `picky-fish%` declares no initialization arguments, any initialization values supplied in `(new picky-fish% ....)` are propagated to the superclass initialization, i.e., to `fish%`. A subclass can supply additional initialization arguments for its superclass in a `super-new` call, and such initialization arguments take precedence over arguments supplied to `new`. For example, the following `size-10-fish%` class always generates fish of size 10:

```
(define size-10-fish% (class fish% (super-new [size 10])))

> (send (new size-10-fish%) get-size)
10
```

In the case of `size-10-fish%`, supplying a size initialization argument with `new` would result in an initialization error; because the size in `super-new` takes precedence, a size supplied to `new` would have no target declaration.

An initialization argument is optional if the `class` form declares a default value. For example, the following `default-10-fish%` class accepts a size initialization argument, but its value defaults to 10 if no value is supplied on instantiation:

```
(define default-10-fish% (class fish%
                           (init [size 10])
                           (super-new [size size])))

> (new default-10-fish%)
(object:default-10-fish% ...)
> (new default-10-fish% [size 20])
(object:default-10-fish% ...)
```

In this example, the `super-new` call propagates its own size value as the size initialization argument to the superclass.

13.3 Internal and External Names

The two uses of `size` in `default-10-fish%` expose the double life of class-member identifiers. When `size` is the first identifier of a bracketed pair in `new` or `super-new`, `size` is an *external name* that is symbolically matched to an initialization argument in a class. When `size` appears as an expression within `default-10-fish%`, `size` is an *internal name* that is lexically scoped. Similarly, a call to an inherited `eat` method uses `eat` as an internal name, whereas a `send` of `eat` uses `eat` as an external name.

The full syntax of the `class` form allows a programmer to specify distinct internal and external names for a class member. Since internal names are local, they can be renamed to avoid shadowing or conflicts. Such renaming is not frequently necessary, but workarounds in the absence of renaming can be especially cumbersome.

13.4 Interfaces

Interfaces are useful for checking that an object or a class implements a set of methods with a particular (implied) behavior. This use of interfaces is helpful even without a static type system (which is the main reason that Java has interfaces).

An interface in Racket is created using the `interface` form, which merely declares the method names required to implement the interface. An interface can extend other interfaces, which means that implementations of the interface automatically implement the extended interfaces.

```
(interface (superinterface-expr ...) id ...)
```

To declare that a class implements an interface, the `class*` form must be used instead of `class`:

```
(class* superclass-expr (interface-expr ...) decl-or-expr ...)
```

For example, instead of forcing all fish classes to be derived from `fish%`, we can define `fish-interface` and change the `fish%` class to declare that it implements `fish-interface`:

```
(define fish-interface (interface () get-size grow eat))
(define fish% (class* object% (fish-interface) ....))
```

If the definition of `fish%` does not include `get-size`, `grow`, and `eat` methods, then an error is signaled in the evaluation of the `class*` form, because implementing the `fish-interface` interface requires those methods.

207

The `is-a?` predicate accepts either a class or interface as its first argument and an object as its second argument. When given a class, `is-a?` checks whether the object is an instance of that class or a derived class. When given an interface, `is-a?` checks whether the object's class implements the interface. In addition, the `implementation?` predicate checks whether a given class implements a given interface.

13.5   Final, Augment, and Inner

As in Java, a method in a `class` form can be specified as *final*, which means that a subclass cannot override the method. A final method is declared using `public-final` or `override-final`, depending on whether the declaration is for a new method or an overriding implementation.

Between the extremes of allowing arbitrary overriding and disallowing overriding entirely, the class system also supports Beta-style *augmentable* methods [Goldberg04]. A method declared with `pubment` is like `public`, but the method cannot be overridden in subclasses; it can be augmented only. A `pubment` method must explicitly invoke an augmentation (if any) using `inner`; a subclass augments the method using `augment`, instead of `override`.

In general, a method can switch between augment and override modes in a class derivation. The `augride` method specification indicates an augmentation to a method where the augmentation is itself overrideable in subclasses (though the superclass's implementation cannot be overridden). Similarly, `overment` overrides a method and makes the overriding implementation augmentable.

13.6   Controlling the Scope of External Names

As noted in §13.3 "Internal and External Names", class members have both internal and external names. A member definition binds an internal name locally, and this binding can be locally renamed. External names, in contrast, have global scope by default, and a member definition does not bind an external name. Instead, a member definition refers to an existing binding for an external name, where the member name is bound to a *member key*; a class ultimately maps member keys to methods, fields, and initialization arguments.

Recall the `hungry-fish%` class expression:

```
(define hungry-fish% (class fish% ....
                       (inherit eat)
                       (define/public (eat-more fish1 fish2)
                         (eat fish1) (eat fish2))))
```

During its evaluation, the `hungry-fish%` and `fish%` classes refer to the same global binding of `eat`. At run time, calls to `eat` in `hungry-fish%` are matched with the `eat` method in

`fish%` through the shared method key that is bound to `eat`.

The default binding for an external name is global, but a programmer can introduce an external-name binding with the `define-member-name` form.

```
(define-member-name id member-key-expr)
```

In particular, by using `(generate-member-key)` as the `member-key-expr`, an external name can be localized for a particular scope, because the generated member key is inaccessible outside the scope. In other words, `define-member-name` gives an external name a kind of package-private scope, but generalized from packages to arbitrary binding scopes in Racket.

For example, the following `fish%` and `pond%` classes cooperate via a `get-depth` method that is only accessible to the cooperating classes:

```
(define-values (fish% pond%) ; two mutually recursive classes
  (let ()
    (define-member-name get-depth (generate-member-key))
    (define fish%
      (class ....
        (define my-depth ....)
        (define my-pond ....)
        (define/public (dive amt)
          (set! my-depth
                (min (+ my-depth amt)
                     (send my-pond get-depth))))))
    (define pond%
      (class ....
        (define current-depth ....)
        (define/public (get-depth) current-depth)))
    (values fish% pond%)))
```

External names are in a namespace that separates them from other Racket names. This separate namespace is implicitly used for the method name in `send`, for initialization-argument names in `new`, or for the external name in a member definition. The special form `member-name-key` provides access to the binding of an external name in an arbitrary expression position: `(member-name-key id)` produces the member-key binding of `id` in the current scope.

A member-key value is primarily used with a `define-member-name` form. Normally, then, `(member-name-key id)` captures the method key of `id` so that it can be communicated to a use of `define-member-name` in a different scope. This capability turns out to be useful for generalizing mixins, as discussed next.

13.7 Mixins

Since `class` is an expression form instead of a top-level declaration as in Smalltalk and Java, a `class` form can be nested inside any lexical scope, including `lambda`. The result is a *mixin*, i.e., a class extension that is parameterized with respect to its superclass.

For example, we can parameterize the `picky-fish%` class over its superclass to define `picky-mixin`:

```
(define (picky-mixin %)
  (class % (super-new)
    (define/override (grow amt) (super grow (* 3/4 amt)))))
(define picky-fish% (picky-mixin fish%))
```

Many small differences between Smalltalk-style classes and Racket classes contribute to the effective use of mixins. In particular, the use of `define/override` makes explicit that `picky-mixin` expects a class with a `grow` method. If `picky-mixin` is applied to a class without a `grow` method, an error is signaled as soon as `picky-mixin` is applied.

Similarly, a use of `inherit` enforces a "method existence" requirement when the mixin is applied:

```
(define (hungry-mixin %)
  (class % (super-new)
    (inherit eat)
    (define/public (eat-more fish1 fish2)
      (eat fish1)
      (eat fish2))))
```

The advantage of mixins is that we can easily combine them to create new classes whose implementation sharing does not fit into a single-inheritance hierarchy—without the ambiguities associated with multiple inheritance. Equipped with `picky-mixin` and `hungry-mixin`, creating a class for a hungry, yet picky fish is straightforward:

```
(define picky-hungry-fish%
  (hungry-mixin (picky-mixin fish%)))
```

The use of keyword initialization arguments is critical for the easy use of mixins. For example, `picky-mixin` and `hungry-mixin` can augment any class with suitable `eat` and `grow` methods, because they do not specify initialization arguments and add none in their `super-new` expressions:

```
(define person%
  (class object%
    (init name age)
    ....
    (define/public (eat food) ....)
```

```
      (define/public (grow amt) ....)))
  (define child% (hungry-mixin (picky-mixin person%)))
  (define oliver (new child% [name "Oliver"] [age 6]))
```

Finally, the use of external names for class members (instead of lexically scoped identifiers) makes mixin use convenient. Applying `picky-mixin` to `person%` works because the names `eat` and `grow` match, without any a priori declaration that `eat` and `grow` should be the same method in `fish%` and `person%`. This feature is a potential drawback when member names collide accidentally; some accidental collisions can be corrected by limiting the scope external names, as discussed in §13.6 "Controlling the Scope of External Names".

13.7.1   Mixins and Interfaces

Using `implementation?`, `picky-mixin` could require that its base class implements `grower-interface`, which could be implemented by both `fish%` and `person%`:

```
  (define grower-interface (interface () grow))
  (define (picky-mixin %)
    (unless (implementation? % grower-interface)
      (error "picky-mixin: not a grower-interface class"))
    (class % ....))
```

Another use of interfaces with a mixin is to tag classes generated by the mixin, so that instances of the mixin can be recognized. In other words, `is-a?` cannot work on a mixin represented as a function, but it can recognize an interface (somewhat like a *specialization interface*) that is consistently implemented by the mixin. For example, classes generated by `picky-mixin` could be tagged with `picky-interface`, enabling the `is-picky?` predicate:

```
  (define picky-interface (interface ()))
  (define (picky-mixin %)
    (unless (implementation? % grower-interface)
      (error "picky-mixin: not a grower-interface class"))
    (class* % (picky-interface) ....))
  (define (is-picky? o)
    (is-a? o picky-interface))
```

13.7.2   The `mixin` Form

To codify the `lambda-plus-class` pattern for implementing mixins, including the use of interfaces for the domain and range of the mixin, the class system provides a `mixin` macro:

```
(mixin (interface-expr ...) (interface-expr ...)
  decl-or-expr ...)
```

The first set of `interface-expr`s determines the domain of the mixin, and the second set
determines the range. That is, the expansion is a function that tests whether a given base class
implements the first sequence of `interface-expr`s and produces a class that implements
the second sequence of `interface-expr`s. Other requirements, such as the presence of
`inherited` methods in the superclass, are then checked for the `class` expansion of the
`mixin` form.

Mixins not only override methods and introduce public methods, they can also augment
methods, introduce augment-only methods, add an overrideable augmentation, and add an
augmentable override — all of the things that a class can do (see §13.5 "Final, Augment,
and Inner").

13.7.3 Parameterized Mixins

As noted in §13.6 "Controlling the Scope of External Names", external names can be bound
with `define-member-name`. This facility allows a mixin to be generalized with respect to
the methods that it defines and uses. For example, we can parameterize `hungry-mixin` with
respect to the external member key for `eat`:

```
(define (make-hungry-mixin eat-method-key)
  (define-member-name eat eat-method-key)
  (mixin () () (super-new)
    (inherit eat)
    (define/public (eat-more x y) (eat x) (eat y))))
```

To obtain a particular hungry-mixin, we must apply this function to a member key that refers
to a suitable `eat` method, which we can obtain using `member-name-key`:

```
((make-hungry-mixin (member-name-key eat))
 (class object% .... (define/public (eat x) 'yum)))
```

Above, we apply `hungry-mixin` to an anonymous class that provides `eat`, but we can also
combine it with a class that provides `chomp`, instead:

```
((make-hungry-mixin (member-name-key chomp))
 (class object% .... (define/public (chomp x) 'yum)))
```

13.8 Traits

A *trait* is similar to a mixin, in that it encapsulates a set of methods to be added to a class. A trait is different from a mixin in that its individual methods can be manipulated with trait operators such as `trait-sum` (merge the methods of two traits), `trait-exclude` (remove a method from a trait), and `trait-alias` (add a copy of a method with a new name; do not redirect any calls to the old name).

The practical difference between mixins and traits is that two traits can be combined, even if they include a common method and even if neither method can sensibly override the other. In that case, the programmer must explicitly resolve the collision, usually by aliasing methods, excluding methods, and merging a new trait that uses the aliases.

Suppose our `fish%` programmer wants to define two class extensions, `spots` and `stripes`, each of which includes a `get-color` method. The fish's spot color should not override the stripe color nor vice versa; instead, a `spots+stripes-fish%` should combine the two colors, which is not possible if `spots` and `stripes` are implemented as plain mixins. If, however, `spots` and `stripes` are implemented as traits, they can be combined. First, we alias `get-color` in each trait to a non-conflicting name. Second, the `get-color` methods are removed from both and the traits with only aliases are merged. Finally, the new trait is used to create a class that introduces its own `get-color` method based on the two aliases, producing the desired `spots+stripes` extension.

13.8.1 Traits as Sets of Mixins

One natural approach to implementing traits in Racket is as a set of mixins, with one mixin per trait method. For example, we might attempt to define the spots and stripes traits as follows, using association lists to represent sets:

```
(define spots-trait
  (list (cons 'get-color
              (lambda (%) (class % (super-new)
                            (define/public (get-color)
                              'black))))))
(define stripes-trait
  (list (cons 'get-color
              (lambda (%) (class % (super-new)
                            (define/public (get-color)
                              'red))))))
```

A set representation, such as the above, allows `trait-sum` and `trait-exclude` as simple manipulations; unfortunately, it does not support the `trait-alias` operator. Although a mixin can be duplicated in the association list, the mixin has a fixed method name, e.g., `get-color`, and mixins do not support a method-rename operation. To support `trait-`

`alias`, we must parameterize the mixins over the external method name in the same way that `eat` was parameterized in §13.7.3 "Parameterized Mixins".

To support the `trait-alias` operation, `spots-trait` should be represented as:

```
(define spots-trait
  (list (cons (member-name-key get-color)
              (lambda (get-color-key %)
                (define-member-name get-color get-color-key)
                (class % (super-new)
                  (define/public (get-color) 'black))))))
```

When the `get-color` method in `spots-trait` is aliased to `get-trait-color` and the `get-color` method is removed, the resulting trait is the same as

```
(list (cons (member-name-key get-trait-color)
            (lambda (get-color-key %)
              (define-member-name get-color get-color-key)
              (class % (super-new)
                (define/public (get-color) 'black)))))
```

To apply a trait T to a class C and obtain a derived class, we use `((trait->mixin T) C)`. The `trait->mixin` function supplies each mixin of T with the key for the mixin's method and a partial extension of C:

```
(define ((trait->mixin T) C)
  (foldr (lambda (m %) ((cdr m) (car m) %)) C T))
```

Thus, when the trait above is combined with other traits and then applied to a class, the use of `get-color` becomes a reference to the external name `get-trait-color`.

13.8.2   Inherit and Super in Traits

This first implementation of traits supports `trait-alias`, and it supports a trait method that calls itself, but it does not support trait methods that call each other. In particular, suppose that a spot-fish's market value depends on the color of its spots:

```
(define spots-trait
  (list (cons (member-name-key get-color) ....)
        (cons (member-name-key get-price)
              (lambda (get-price %) ....
                (class % ....
                  (define/public (get-price)
                    .... (get-color) ....))))))
```

In this case, the definition of `spots-trait` fails, because `get-color` is not in scope for the `get-price` mixin. Indeed, depending on the order of mixin application when the trait is applied to a class, the `get-color` method may not be available when `get-price` mixin is applied to the class. Therefore adding an `(inherit get-color)` declaration to the `get-price` mixin does not solve the problem.

One solution is to require the use of `(send this get-color)` in methods such as `get-price`. This change works because `send` always delays the method lookup until the method call is evaluated. The delayed lookup is more expensive than a direct call, however. Worse, it also delays checking whether a `get-color` method even exists.

A second, effective, and efficient solution is to change the encoding of traits. Specifically, we represent each method as a pair of mixins: one that introduces the method and one that implements it. When a trait is applied to a class, all of the method-introducing mixins are applied first. Then the method-implementing mixins can use `inherit` to directly access any introduced method.

```
(define spots-trait
  (list (list (local-member-name-key get-color)
              (lambda (get-color get-price %) ....
                (class % ....
                  (define/public (get-color) (void))))
              (lambda (get-color get-price %) ....
                (class % ....
                  (define/override (get-color) 'black))))
        (list (local-member-name-key get-price)
              (lambda (get-price get-color %) ....
                (class % ....
                  (define/public (get-price) (void))))
              (lambda (get-color get-price %) ....
                (class % ....
                  (inherit get-color)
                  (define/override (get-price)
                    .... (get-color) ....))))))
```

With this trait encoding, `trait-alias` adds a new method with a new name, but it does not change any references to the old method.

13.8.3 The `trait` Form

The general-purpose trait pattern is clearly too complex for a programmer to use directly, but it is easily codified in a `trait` macro:

```
(trait trait-clause ...)
```

The `ids` in the optional `inherit` clause are available for direct reference in the method `exprs`, and they must be supplied either by other traits or the base class to which the trait is ultimately applied.

Using this form in conjunction with trait operators such as `trait-sum`, `trait-exclude`, `trait-alias`, and `trait->mixin`, we can implement `spots-trait` and `stripes-trait` as desired.

```
(define spots-trait
  (trait
    (define/public (get-color) 'black)
    (define/public (get-price) ... (get-color) ...)))

(define stripes-trait
  (trait
    (define/public (get-color) 'red)))

(define spots+stripes-trait
  (trait-sum
   (trait-exclude (trait-alias spots-trait
                               get-color get-spots-color)
                  get-color)
   (trait-exclude (trait-alias stripes-trait
                               get-color get-stripes-color)
                  get-color)
   (trait
     (inherit get-spots-color get-stripes-color)
     (define/public (get-color)
       .... (get-spots-color) .... (get-stripes-color) ....))))
```

13.9   Class Contracts

As classes are values, they can flow across contract boundaries, and we may wish to protect parts of a given class with contracts. For this, the `class/c` form is used. The `class/c` form has many subforms, which describe two types of contracts on fields and methods: those that affect uses via instantiated objects and those that affect subclasses.

13.9.1   External Class Contracts

In its simplest form, `class/c` protects the public fields and methods of objects instantiated from the contracted class. There is also an `object/c` form that can be used to similarly protect the public fields and methods of a particular object. Take the following definition of `animal%`, which uses a public field for its `size` attribute:

```
(define animal%
  (class object%
    (super-new)
    (field [size 10])
    (define/public (eat food)
      (set! size (+ size (get-field size food))))))
```

For any instantiated `animal%`, accessing the `size` field should return a positive number. Also, if the `size` field is set, it should be assigned a positive number. Finally, the `eat` method should receive an argument which is an object with a `size` field that contains a positive number. To ensure these conditions, we will define the `animal%` class with an appropriate contract:

```
(define positive/c (and/c number? positive?))
(define edible/c (object/c (field [size positive/c])))
(define/contract animal%
  (class/c (field [size positive/c])
           [eat (->m edible/c void?)])
  (class object%
    (super-new)
    (field [size 10])
    (define/public (eat food)
      (set! size (+ size (get-field size food))))))
```

Here we use `->m` to describe the behavior of `eat` since we do not need to describe any requirements for the `this` parameter. Now that we have our contracted class, we can see that the contracts on both `size` and `eat` are enforced:

```
> (define bob (new animal%))
> (set-field! size bob 3)
> (get-field size bob)
3
> (set-field! size bob 'large)
```
contract violation: expected <positive/c>, given: 'large
  contract on animal% from (definition animal%), blaming
top-level
  contract:
    (class/c
     (eat
       (->
        (object/c (field (size positive/c)))
        void?))
     (field (size positive/c)))
  at: eval:22.0
```
> (define richie (new animal%))
> (send bob eat richie)
```

```
> (get-field size bob)
13
> (define rock (new object%))
> (send bob eat rock)
```
contract violation: no public field size
   contract on animal% from (definition animal%), blaming
top-level
   contract:
      (class/c
       (eat
         (->
         (object/c (field (size positive/c)))
         void?))
       (field (size positive/c)))
   at: eval:22.0
```
> (define giant (new (class object% (super-new) (field [size 'large])))))
> (send bob eat giant)
```
contract violation: expected <positive/c>, given: 'large
   contract on animal% from (definition animal%), blaming
top-level
   contract:
      (class/c
       (eat
         (->
         (object/c (field (size positive/c)))
         void?))
       (field (size positive/c)))
   at: eval:22.0

There are two important caveats for external class contracts. First, external method contracts are only enforced when the target of dynamic dispatch is the method implementation of the contracted class, which lies within the contract boundary. Overriding that implementation, and thus changing the target of dynamic dispatch, will mean that the contract is no longer enforced for clients, since accessing the method no longer crosses the contract boundary. Unlike external method contracts, external field contracts are always enforced for clients of subclasses, since fields cannot be overridden or shadowed.

Second, these contracts do not restrict subclasses of animal% in any way. Fields and methods that are inherited and used by subclasses are not checked by these contracts, and uses of the superclass's methods via super are also unchecked. The following example illustrates both caveats:

```
(define large-animal%
  (class animal%
    (super-new)
    (inherit-field size)
```

218

```
      (set! size 'large)
      (define/override (eat food)
        (display "Nom nom nom") (newline))))

> (define elephant (new large-animal%))
> (send elephant eat (new object%))
Nom nom nom
> (get-field size elephant)
```

self-contract violation: expected <positive/c>, given:
'large
  contract on animal% from (definition animal%), blaming
(definition animal%)
  contract:
    (class/c
    (eat
      (->
      (object/c (field (size positive/c)))
      void?))
    (field (size positive/c)))
  at: eval:22.0

13.9.2   Internal Class Contracts

Notice that retrieving the `size` field from the object `elephant` blames `animal%` for the contract violation. This blame is correct, but unfair to the `animal%` class, as we have not yet provided it with a method for protecting itself from subclasses. To this end we add internal class contracts, which provide directives to subclasses for how they may access and override features of the superclass. This distinction between external and internal class contracts allows for weaker contracts within the class hierarchy, where invariants may be broken internally by subclasses but should be enforced for external uses via instantiated objects.

As a simple example of what kinds of protection are available, we provide an example aimed at the `animal%` class that uses all the applicable forms:

```
(class/c (field [size positive/c])
         (inherit-field [size positive/c])
         [eat (->m edible/c void?)]
         (inherit [eat (->m edible/c void?)])
         (super [eat (->m edible/c void?)])
         (override [eat (->m edible/c void?)]))
```

This class contract not only ensures that objects of class `animal%` are protected as before, but also ensure that subclasses of `animal%` only store appropriate values within the `size` field and use the implementation of `size` from `animal%` appropriately. These contract forms

only affect uses within the class hierarchy, and only for method calls that cross the contract boundary.

That means that `inherit` will only affect subclass uses of a method until a subclass overrides that method, and that `override` only affects calls from the superclass into a subclass's overriding implementation of that method. Since these only affect internal uses, the `override` form does not automatically enter subclasses into obligations when objects of those classes are used. Also, use of `override` only makes sense, and thus can only be used, for methods where no Beta-style augmentation has taken place. The following example shows this difference:

```
(define/contract sloppy-eater%
  (class/c [eat (->m edible/c edible/c)])
  (begin
    (define/contract glutton%
      (class/c (override [eat (->m edible/c void?)]))
      (class animal%
        (super-new)
        (inherit eat)
        (define/public (gulp food-list)
          (for ([f food-list])
            (eat f)))))
    (class glutton%
      (super-new)
      (inherit-field size)
      (define/override (eat f)
        (let ([food-size (get-field size f)])
          (set! size (/ food-size 2))
          (set-field! size f (/ food-size 2))
          f)))))

> (define pig (new sloppy-eater%))
> (define slop1 (new animal%))
> (define slop2 (new animal%))
> (define slop3 (new animal%))
> (send pig eat slop1)
(object:animal% ...)
> (get-field size slop1)
5
> (send pig gulp (list slop1 slop2 slop3))
contract violation: expected <void?>, given:
(object:animal% ...)
  contract on glutton% from (definition glutton%), blaming
(definition sloppy-eater%)
  contract:
    (class/c
```

```
(override (eat
             (->
              (object/c
                (field (size positive/c)))
              void?))))
at: eval:38.0
```

In addition to the internal class contract forms shown here, there are similar forms for Beta-style augmentable methods. The `inner` form describes to the subclass what is expected from augmentations of a given method. Both `augment` and `augride` tell the subclass that the given method is a method which has been augmented and that any calls to the method in the subclass will dynamically dispatch to the appropriate implementation in the superclass. Such calls will be checked according to the given contract. The two forms differ in that use of `augment` signifies that subclasses can augment the given method, whereas use of `augride` signifies that subclasses must override the current augmentation instead.

This means that not all forms can be used at the same time. Only one of the `override`, `augment`, and `augride` forms can be used for a given method, and none of these forms can be used if the given method has been finalized. In addition, `super` can be specified for a given method only if `augride` or `override` can be specified. Similarly, `inner` can be specified only if `augment` or `augride` can be specified.

14 Units (Components)

Units organize a program into separately compilable and reusable *components*. A unit resembles a procedure in that both are first-class values that are used for abstraction. While procedures abstract over values in expressions, units abstract over names in collections of definitions. Just as a procedure is called to evaluate its expressions given actual arguments for its formal parameters, a unit is *invoked* to evaluate its definitions given actual references for its imported variables. Unlike a procedure, however, a unit's imported variables can be partially linked with the exported variables of another unit *prior to invocation*. Linking merges multiple units together into a single compound unit. The compound unit itself imports variables that will be propagated to unresolved imported variables in the linked units, and re-exports some variables from the linked units for further linking.

14.1 Signatures and Units

The interface of a unit is described in terms of *signatures*. Each signature is defined (normally within a `module`) using `define-signature`. For example, the following signature, placed in a `"toy-factory-sig.rkt"` file, describes the exports of a component that implements a toy factory:

By convention, signature names with ^.

`"toy-factory-sig.rkt"`

```racket
#lang racket

(define-signature toy-factory^
  (build-toys  ; (integer? -> (listof toy?))
   repaint     ; (toy? symbol? -> toy?)
   toy?        ; (any/c -> boolean?)
   toy-color)) ; (toy? -> symbol?)

(provide toy-factory^)
```

An implementation of the `toy-factory^` signature is written using `define-unit` with an export clause that names `toy-factory^`:

By convention, unit names with @.

`"simple-factory-unit.rkt"`

```racket
#lang racket

(require "toy-factory-sig.rkt")

(define-unit simple-factory@
  (import)
  (export toy-factory^)

  (printf "Factory started.\n")
```

222

```
    (define-struct toy (color) #:transparent)

    (define (build-toys n)
      (for/list ([i (in-range n)])
        (make-toy 'blue)))

    (define (repaint t col)
      (make-toy col)))

  (provide simple-factory@)
```

The `toy-factory^` signature also could be referenced by a unit that needs a toy factory to implement something else. In that case, `toy-factory^` would be named in an `import` clause. For example, a toy store would get toys from a toy factory. (Suppose, for the sake of an example with interesting features, that the store is willing to sell only toys in a particular color.)

"toy-store-sig.rkt"

```
 #lang racket

 (define-signature toy-store^
   (store-color      ; (-> symbol?)
    stock!           ; (integer? -> void?)
    get-inventory)) ; (-> (listof toy?))

 (provide toy-store^)
```

"toy-store-unit.rkt"

```
 #lang racket

 (require "toy-store-sig.rkt"
          "toy-factory-sig.rkt")

 (define-unit toy-store@
   (import toy-factory^)
   (export toy-store^)

   (define inventory null)

   (define (store-color) 'green)

   (define (maybe-repaint t)
     (if (eq? (toy-color t) (store-color))
         t
         (repaint t (store-color)))))
```

```
  (define (stock! n)
    (set! inventory
          (append inventory
                  (map maybe-repaint
                       (build-toys n)))))

  (define (get-inventory) inventory))

(provide toy-store@)
```

Note that "toy-store-unit.rkt" imports "toy-factory-sig.rkt", but not "simple-factory-unit.rkt". Consequently, the toy-store@ unit relies only on the specification of a toy factory, not on a specific implementation.

14.2   Invoking Units

The simple-factory@ unit has no imports, so it can be invoked directly using invoke-unit:

```
> (require "simple-factory-unit.rkt")
> (invoke-unit simple-factory@)
Factory started.
```

The invoke-unit form does not make the body definitions available, however, so we cannot build any toys with this factory. The define-values/invoke-unit form binds the identifiers of a signature to the values supplied by a unit (to be invoked) that implements the signature:

```
> (define-values/invoke-unit/infer simple-factory@)
Factory started.
> (build-toys 3)
(list  (toy 'blue) (toy 'blue) (toy 'blue))
```

Since simple-factory@ exports the toy-factory^ signature, each identifier in toy-factory^ is defined by the define-values/invoke-unit/infer form. The /infer part of the form name indicates that the identifiers bound by the declaration are inferred from simple-factory@.

Now that the identifiers in toy-factory^ are defined, we can also invoke toy-store@, which imports toy-factory^ to produce toy-store^:

```
> (require "toy-store-unit.rkt")
> (define-values/invoke-unit/infer toy-store@)
> (get-inventory)
```

```
'()
> (stock! 2)
> (get-inventory)
(list  (toy 'green) (toy 'green))
```

Again, the /infer part define-values/invoke-unit/infer determines that toy-store@ imports toy-factory^, and so it supplies the top-level bindings that match the names in toy-factory^ as imports to toy-store@.

14.3   Linking Units

We can make our toy economy more efficient by having toy factories that cooperate with stores, creating toys that do not have to be repainted. Instead, the toys are always created using the store's color, which the factory gets by importing toy-store^:

"store-specific-factory-unit.rkt"

```
#lang racket

(require "toy-factory-sig.rkt")

(define-unit store-specific-factory@
  (import toy-store^)
  (export toy-factory^)

  (define-struct toy () #:transparent)

  (define (toy-color t) (store-color))

  (define (build-toys n)
    (for/list ([i (in-range n)])
      (make-toy)))

  (define (repaint t col)
    (error "cannot repaint")))

(provide store-specific-factory@)
```

To invoke store-specific-factory@, we need toy-store^ bindings to supply to the unit. But to get toy-store^ bindings by invoking toy-store@, we will need a toy factory! The unit implementations are mutually dependent, and we cannot invoke either before the other.

The solution is to *link* the units together, and then we can invoke the combined units. The define-compound-unit/infer form links any number of units to form a combined unit. It can propagate imports and exports from the linked units, and it can satisfy each unit's

225

imports using the exports of other linked units.

```
> (require "store-specific-factory-unit.rkt")
> (define-compound-unit/infer toy-store+factory@
    (import)
    (export toy-factory^ toy-store^)
    (link store-specific-factory@
          toy-store@))
```

The overall result above is a unit `toy-store+factory@` that exports both `toy-factory^`
and `toy-store^`. The connection between `store-specific-factory@` and `toy-store@`
is inferred from the signatures that each imports and exports.

This unit has no imports, so we can always invoke it:

```
> (define-values/invoke-unit/infer toy-store+factory@)
> (stock! 2)
> (get-inventory)
(list  (toy )(toy ))
> (map toy-color (get-inventory))
'(green green)
```

14.4   First-Class Units

The `define-unit` form combines `define` with a `unit` form, similar to the way that `(de-
fine (f x) ....)` combines `define` followed by an identifier with an implicit `lambda`.

Expanding the shorthand, the definition of `toy-store@` could almost be written as

```
(define toy-store@
  (unit
   (import toy-factory^)
   (export toy-store^)

   (define inventory null)

   (define (store-color) 'green)
   ....))
```

A difference between this expansion and `define-unit` is that the imports and exports of
`toy-store@` cannot be inferred. That is, besides combining `define` and `unit`, `define-
unit` attaches static information to the defined identifier so that its signature information is
available statically to `define-values/invoke-unit/infer` and other forms.

Despite the drawback of losing static signature information, `unit` can be useful in combina-

tion with other forms that work with first-class values. For example, we could wrap a `unit` that creates a toy store in a `lambda` to supply the store's color:

"toy-store-maker.rkt"

```
#lang racket

(require "toy-store-sig.rkt"
         "toy-factory-sig.rkt")

(define toy-store@-maker
  (lambda (the-color)
   (unit
    (import toy-factory^)
    (export toy-store^)

    (define inventory null)

    (define (store-color) the-color)

    ; the rest is the same as before

    (define (maybe-repaint t)
      (if (eq? (toy-color t) (store-color))
          t
          (repaint t (store-color))))

    (define (stock! n)
      (set! inventory
            (append inventory
                    (map maybe-repaint
                         (build-toys n)))))

    (define (get-inventory) inventory))))

(provide toy-store@-maker)
```

To invoke a unit created by `toy-store@-maker`, we must use `define-values/invoke-unit`, instead of the `/infer` variant:

```
> (require "simple-factory-unit.rkt")
> (define-values/invoke-unit/infer simple-factory@)
Factory started.
> (require "toy-store-maker.rkt")
> (define-values/invoke-unit (toy-store@-maker 'purple)
    (import toy-factory^)
    (export toy-store^))
```

```
> (stock! 2)
> (get-inventory)
(list  (toy 'purple) (toy 'purple))
```

In the `define-values/invoke-unit` form, the (import `toy-factory^`) line takes
bindings from the current context that match the names in `toy-factory^` (the ones that
we created by invoking `simple-factory@`), and it supplies them as imports to `toy-`
`store@`. The (export `toy-store^`) clause indicates that the unit produced by `toy-`
`store@-maker` will export `toy-store^`, and the names from that signature are defined
after invoking the unit.

To link a unit from `toy-store@-maker`, we can use the `compound-unit` form:

```
> (require "store-specific-factory-unit.rkt")
> (define toy-store+factory@
    (compound-unit
     (import)
     (export TF TS)
     (link [((TF : toy-factory^)) store-specific-factory@ TS]
           [((TS : toy-store^)) toy-store@ TF])))
```

This `compound-unit` form packs a lot of information into one place. The left-hand-
side TF and TS in the `link` clause are binding identifiers. The identifier TF is essentially
bound to the elements of `toy-factory^` as implemented by `store-specific-factory@`.
The identifier TS is similarly bound to the elements of `toy-store^` as implemented by
`toy-store@`. Meanwhile, the elements bound to TS are supplied as imports for `store-`
`specific-factory@`, since TS follows `store-specific-factory@`. The elements bound
to TF are similarly supplied to `toy-store@`. Finally, (export TF TS) indicates that the
elements bound to TF and TS are exported from the compound unit.

The above `compound-unit` form uses `store-specific-factory@` as a first-class unit,
even though its information could be inferred. Every unit can be used as a first-class unit, in
addition to its use in inference contexts. Also, various forms let a programmer bridge the gap
between inferred and first-class worlds. For example, `define-unit-binding` binds a new
identifier to the unit produced by an arbitrary expression; it statically associates signature
information to the identifier, and it dynamically checks the signatures against the first-class
unit produced by the expression.

14.5   Whole-`module` Signatures and Units

In programs that use units, modules like "toy-factory-sig.rkt" and "simple-
factory-unit.rkt" are common. The `racket/signature` and `racket/unit` module
names can be used as languages to avoid much of the boilerplate module, signature, and unit
declaration text.

For example, "toy-factory-sig.rkt" can be written as

```racket
#lang racket/signature

build-toys  ; (integer? -> (listof toy?))
repaint     ; (toy? symbol? -> toy?)
toy?        ; (any/c -> boolean?)
toy-color   ; (toy? -> symbol?)
```

The signature `toy-factory^` is automatically provided from the module, inferred from the filename "toy-factory-sig.rkt" by replacing the "-sig.rkt" suffix with `^`.

Similarly, "simple-factory-unit.rkt" module can be written

```racket
#lang racket/unit

(require "toy-factory-sig.rkt")

(import)
(export toy-factory^)

(printf "Factory started.\n")

(define-struct toy (color) #:transparent)

(define (build-toys n)
  (for/list ([i (in-range n)])
    (make-toy 'blue)))

(define (repaint t col)
  (make-toy col))
```

The unit `simple-factory@` is automatically provided from the module, inferred from the filename "simple-factory-unit.rkt" by replacing the "-unit.rkt" suffix with `@`.

14.6   Contracts for Units

There are a couple of ways of protecting units with contracts. One way is useful when writing new signatures, and the other handles the case when a unit must conform to an already existing signature.

14.6.1 Adding Contracts to Signatures

When contracts are added to a signature, then all units which implement that signature are protected by those contracts. The following version of the `toy-factory^` signature adds the contracts previously written in comments:

"contracted-toy-factory-sig.rkt"

```racket
#lang racket

(define-signature contracted-toy-factory^
  ((contracted
    [build-toys (-> integer? (listof toy?))]
    [repaint    (-> toy? symbol? toy?)]
    [toy?       (-> any/c boolean?)]
    [toy-color  (-> toy? symbol?)]))))

(provide contracted-toy-factory^)
```

Now we take the previous implementation of `simple-factory@` and implement this version of `toy-factory^` instead:

"contracted-simple-factory-unit.rkt"

```racket
#lang racket

(require "contracted-toy-factory-sig.rkt")

(define-unit contracted-simple-factory@
  (import)
  (export contracted-toy-factory^)

  (printf "Factory started.\n")

  (define-struct toy (color) #:transparent)

  (define (build-toys n)
    (for/list ([i (in-range n)])
      (make-toy 'blue)))

  (define (repaint t col)
    (make-toy col)))

(provide contracted-simple-factory@)
```

As before, we can invoke our new unit and bind the exports so that we can use them. This time, however, misusing the exports causes the appropriate contract errors.

```
> (require "contracted-simple-factory-unit.rkt")
> (define-values/invoke-unit/infer contracted-simple-factory@)
Factory started.
> (build-toys 3)
(list  (toy 'blue) (toy 'blue) (toy 'blue))
> (build-toys #f)
```
contract violation: expected <integer?>, given: #f
   contract on build-toys from
     (unit contracted-simple-factory@)
   blaming top-level
   contract: (-> integer? (listof toy?))
       at: eval:32.0
```
> (repaint 3 'blue)
```
contract violation: expected <toy?>, given: 3
   contract on repaint from
     (unit contracted-simple-factory@)
   blaming top-level
   contract: (-> toy? symbol? toy?)
       at: eval:32.0

14.6.2 Adding Contracts to Units

However, sometimes we may have a unit that must conform to an already existing signature
that is not contracted. In this case, we can create a unit contract with unit/c or use the
define-unit/contract form, which defines a unit which has been wrapped with a unit
contract.

For example, here's a version of toy-factory@ which still implements the regular toy-
factory^, but whose exports have been protected with an appropriate unit contract.

> "wrapped-simple-factory-unit.rkt"

```
#lang racket

(require "toy-factory-sig.rkt")

(define-unit/contract wrapped-simple-factory@
  (import)
  (export (toy-factory^
            [build-toys (-> integer? (listof toy?))]
            [repaint    (-> toy? symbol? toy?)]
            [toy?       (-> any/c boolean?)]
            [toy-color  (-> toy? symbol?)]]))

  (printf "Factory started.\n")
```

```
  (define-struct toy (color) #:transparent)

  (define (build-toys n)
    (for/list ([i (in-range n)])
      (make-toy 'blue)))

  (define (repaint t col)
    (make-toy col)))

(provide contracted-simple-factory@)

> (require "wrapped-simple-factory-unit.rkt")
> (define-values/invoke-unit/infer wrapped-simple-factory@)
Factory started.
> (build-toys 3)
(list  (toy 'blue) (toy 'blue) (toy 'blue))
> (build-toys #f)
```

contract violation: expected <integer?>, given: #f
  contract on wrapped-simple-factory@ from
    (unit wrapped-simple-factory@)
  blaming top-level
  contract:
    (unit/c
     (import)
     *(export (toy-factory^*
                (build-toys
                 (-> integer? (listof toy?)))
                (repaint (-> toy? symbol? toy?))
                (toy? (-> any/c boolean?))
                (toy-color (-> toy? symbol?)))))
  at: <collects>/mzlib/unit.rkt

```
> (repaint 3 'blue)
```

contract violation: expected <toy?>, given: 3
  contract on wrapped-simple-factory@ from
    (unit wrapped-simple-factory@)
  blaming top-level
  contract:
    (unit/c
     (import)
     *(export (toy-factory^*
                (build-toys
                 (-> integer? (listof toy?)))
                (repaint (-> toy? symbol? toy?))
                (toy? (-> any/c boolean?))
                (toy-color (-> toy? symbol?)))))

232

14.7 `unit` **versus** `module`

As a form for modularity, `unit` complements `module`:

- The `module` form is primarily for managing a universal namespace. For example, it allows a code fragment to refer specifically to the `car` operation from `racket/base`—the one that extracts the first element of an instance of the built-in pair datatype—as opposed to any number of other functions with the name `car`. In other word, the `module` construct lets you refer to *the* binding that you want.

- The `unit` form is for parameterizing a code fragment with respect to most any kind of run-time value. For example, it allows a code fragment for work with a `car` function that accepts a single argument, where the specific function is determined later by linking the fragment to another. In other words, the `unit` construct lets you refer to *a* binding that meets some specification.

The `lambda` and `class` forms, among others, also allow paremetrization of code with respect to values that are chosen later. In principle, any of those could be implemented in terms of any of the others. In practice, each form offers certain conveniences—such as allowing overriding of methods or especially simple application to values—that make them suitable for different purposes.

The `module` form is more fundamental than the others, in a sense. After all, a program fragment cannot reliably refer to `lambda`, `class`, or `unit` form without the namespace management provided by `module`. At the same time, because namespace management is closely related to separate expansion and compilation, `module` boundaries end up as separate-compilation boundaries in a way that prohibits mutual dependencies among fragments. For similar reasons, `module` does not separate interface from implementation.

Use `unit` when `module` by itself almost works, but when separately compiled pieces must refer to each other, or when you want a stronger separation between *interface* (i.e., the parts that need to be known at expansion and compilation time) and *implementation* (i.e., the run-time parts). More generally, use `unit` when you need to parameterize code over functions, datatypes, and classes, and when the parameterized code itself provides definitions to be linked with other parameterized code.

15 Reflection and Dynamic Evaluation

Racket is a *dynamic* language. It offers numerous facilities for loading, compiling, and even constructing new code at run time.

15.1 `eval`

The `eval` function takes a "quoted" expression or definition and evaluates it:

```
> (eval '(+ 1 2))
3
```

The power of `eval` that is that an expression can be constructed dynamically:

```
> (define (eval-formula formula)
    (eval '(let ([x 2]
                 [y 3])
             ,formula)))
> (eval-formula '(+ x y))
5
> (eval-formula '(+ (* x y) y))
9
```

Of course, if we just wanted to evaluate expressions with given values for `x` and `y`, we do not need `eval`. A more direct approach is to use first-class functions:

```
> (define (apply-formula formula-proc)
    (formula-proc 2 3))
> (apply-formula (lambda (x y) (+ x y)))
5
> (apply-formula (lambda (x y) (+ (* x y) y)))
9
```

However, if expressions like `(+ x y)` and `(+ (* x y) y)` are read from a file supplied by a user, for example, then `eval` might be appropriate. Simialrly, the REPL reads expressions that are typed by a user and uses `eval` to evaluate them.

Also, `eval` is often used directly or indirectly on whole modules. For example, a program might load a module on demand using `dynamic-require`, which is essentially a wrapper around `eval` to dynamically load the module code.

15.1.1 Local Scopes

The `eval` function cannot see local bindings in the context where it is called. For example, calling `eval` inside an unquoted `let` form to evaluate a formula does not make values visible for `x` and `y`:

```
> (define (broken-eval-formula formula)
    (let ([x 2]
          [y 3])
      (eval formula)))
> (broken-eval-formula '(+ x y))
reference to undefined identifier: x
```

The `eval` function cannot see the `x` and `y` bindings precisely because it is a function, and Racket is a lexically scoped language. Imagine if `eval` were implemented as

```
(define (eval x)
  (eval-expanded (macro-expand x)))
```

then at the point when `eval-expanded` is called, the most recent binding of `x` is to the expression to evaluate, not the `let` binding in `broken-eval-formula`. Lexical scope prevents such confusing and fragile behavior, and consequently prevents `eval` from seeing local bindings in the context where it is called.

You might imagine that even though `eval` cannot see the local bindings in `broken-eval-formula`, there must actually be a data structure mapping `x` to `2` and `y` to `3`, and you would like a way to get that data structure. In fact, no such data structure exists; the compiler is free to replace every use of `x` with `2` at compile time, so that the local binding of `x` does not exist in any concrete sense at run-time. Even when variables cannot be eliminated by constant-folding, normally the names of the variables can be eliminated, and the data structures that hold local values do not resemble a mapping from names to values.

15.1.2 Namespaces

Since `eval` cannot see the bindings from the context where it is called, another mechanism is needed to determine dynamically available bindings. A *namespace* is a first-class value that encapsulates the bindings available for dynamic evaluation.

Some functions, such as `eval`, accept an optional namespace argument. More often, the namespace used by a dynamic operation is the *current namespace* as determined by the `current-namespace` parameter.

When `eval` is used in a REPL, the current namespace is the one that the REPL uses for evaluating expressions. That's why the following interaction successfully accesses `x` via `eval`:

Informally, the term *namespace* is sometimes used interchangeably with *environment* or *scope*. In Racket, the term *namespace* has the more specific, dynamic meaning given above, and it should not be confused with static lexical concepts.

```
> (define x 3)
> (eval 'x)
3
```

In contrast, try the following a simple module and running in directly in DrRacket or supplying the file as a command-line argument to `racket`:

```
#lang racket

(eval '(cons 1 2))
```

This fails because the initial current namespace is empty. When you run `racket` in interactive mode (see §19.1.1 "Interactive Mode"), the initial namespace is initialized with the exports of the `racket` module, but when you run a module directly, the initial namespace starts empty.

In general, it's a bad idea to use `eval` with whatever namespace happens to be installed. Instead, create a namespace explicitly and install it for the call to eval:

```
#lang racket

(define ns (make-base-namespace))
(eval '(cons 1 2) ns) ; works
```

The `make-base-namespace` function creates a namespace that is initialized with the exports of `racket/base`. The later section §15.2 "Manipulating Namespaces" provides more information on creating and configuring namespaces.

15.1.3   Namespaces and Modules

As with `let` bindings, lexical scope means that `eval` cannot automatically see the definitions of a `module` in which it is called. Unlike `let` bindings, however, Racket provides a way to reflect a module into a namespace.

The `module->namespace` function takes a quoted module path and produces a namespace for evaluating expressions and definitions as if they appeared in the `module` body:

```
> (module m racket/base
    (define x 11))
> (require 'm)
> (define ns (module->namespace ''m))
> (eval 'x ns)
11
```

The `module->namespace` function is mostly useful from outside a module, where the module's full name is known. Inside a `module` form, however, the full name of a module may

The double quoting in `''m` is because `'m` is a module path that refers to an interactively declared module, and so `''m` is the quoted form of the path.

236

not be known, because it may depend on where the module source is location when it is eventually loaded.

From within a `module`, use `define-namespace-anchor` to declare a reflection hook on the module, and use `namespace-anchor->namespace` to reel in the module's namespace:

```racket
#lang racket

(define-namespace-anchor a)
(define ns (namespace-anchor->namespace a))

(define x 1)
(define y 2)

(eval '(cons x y) ns) ; produces (1 . 2)
```

15.2 Manipulating Namespaces

A namespace encapsulates two pieces of information:

- A mapping from identifiers to bindings. For example, a namespace might map the identifier `lambda` to the `lambda` form. An "empty" namespace is one that maps every identifier to an uninitialized top-level variable.

- A mapping from module names to module declarations and instances.

The first mapping is used for evaluating expressions in a top-level context, as in `(eval '(lambda (x) (+ x 1)))`. The second mapping is used, for example, by `dynamic-require` to locate a module. The call `(eval '(require racket/base))` normally uses both pieces: the identifier mapping determines the binding of `require`; if it turns out to mean `require`, then the module mapping is used to locate the `racket/base` module.

From the perspective of the core Racket run-time system, all evaluation is reflective. Execution starts with an initial namespace that contains a few primitive modules, and that is further populated by loading files and modules as specified on the command line or as supplied in the REPL. Top-level `require` and `define` forms adjusts the identifier mapping, and module declarations (typically loaded on demand for a `require` form) adjust the module mapping.

15.2.1 Creating and Installing Namespaces

The function `make-empty-namespace` creates a new, empty namespace. Since the namespace is truly empty, it cannot at first be used to evaluate any top-level expression—not even `(require racket)`. In particular,

237

```
(parameterize ([current-namespace (make-empty-namespace)])
  (namespace-require 'racket))
```

fails, because the namespace does not include the primitive modules on which `racket` is built.

To make a namespace useful, some modules must be *attached* from an existing namespace. Attaching a module adjusts the mapping of module names to instances by transitively copying entries (the module and all its imports) from an existing namespace's mapping. Normally, instead of just attaching the primitive modules—whose names and organization are subject to change—a higher-level module is attached, such as `racket` or `racket/base`.

The `make-base-empty-namespace` function provides a namespace that is empty, except that `racket/base` is attached. The resulting namespace is still "empty" in the sense that the identifiers-to-bindings part of the namespace has no mappings; only the module mapping has been populated. Nevertheless, with an initial module mapping, further modules can be loaded.

A namespace created with `make-base-empty-namespace` is suitable for many basic dynamic tasks. For example, suppose that a `my-dsl` library implements a domain-specific language in which you want to execute commands from a user-specified file. A namespace created with `make-base-empty-namespace` is enough to get started:

```
(define (run-dsl file)
  (parameterize ([current-namespace (make-base-empty-namespace)])
    (namespace-require 'my-dsl)
    (load file)))
```

Note that the `parameterize` of `current-namespace` does not affect the meaning of identifiers like `namespace-require` within the `parameterize` body. Those identifiers obtain their meaning from the enclosing context (probably a module). Only expressions that are dynamic with respect to this code, such as the content of `load`ed files, are affected by the `parameterize`.

Another subtle point in the above example is the use of `(namespace-require 'my-dsl)` instead of `(eval '(require my-dsl))`. The latter would not work, because `eval` needs to obtain a meaning for `require` in the namespace, and the namespace's identifier mapping is initially empty. The `namespace-require` function, in contrast, directly imports the given module into the current namespace. Starting with `(namespace-require 'racket/base)` would introduce a binding for `require` and make a subsequent `(eval '(require my-dsl))` work. The above is better, not only because it is more compact, but also because it avoids introducing bindings that are not part of the domain-specific languages.

15.2.2 Sharing Data and Code Across Namespaces

Modules not attached to a new namespace will be loaded and instantiated afresh if they are demanded by evaluation. For example, `racket/base` does not include `racket/class`, and loading `racket/class` again will create a distinct class datatype:

```
> (require racket/class)
> (class? object%)
#t
> (class?
    (parameterize ([current-namespace (make-base-empty-namespace)])
      (namespace-require 'racket/class) ; loads again
      (eval 'object%)))
#f
```

For cases when dynamically loaded code needs to share more code and data with its context, use the `namespace-attach-module` function. The first argument to `namespace-attach-module` is a source namespace from which to draw a module instance; in some cases, the current namespace is known to include the module that needs to be shared:

```
> (require racket/class)
> (class?
    (let ([ns (make-base-empty-namespace)])
      (namespace-attach-module (current-namespace)
                               'racket/class
                               ns)
      (parameterize ([current-namespace ns])
        (namespace-require 'racket/class) ; uses attached
        (eval 'object%))))
#t
```

Within a module, however, the combination of `define-namespace-anchor` and `namespace-anchor->empty-namespace` offers a more reliable method for obtaining a source namespace:

```
#lang racket/base

(require racket/class)

(define-namespace-anchor a)

(define (load-plug-in file)
  (let ([ns (make-base-empty-namespace)])
    (namespace-attach-module (namespace-anchor->empty-namespace a)
                             'racket/class
                             ns)
```

```
(parameterize ([current-namespace ns])
  (dynamic-require file 'plug-in%))))
```

The anchor bound by `namespace-attach-module` connects the run time of a module with the namespace in which a module is loaded (which might differ from the current namespace). In the above example, since the enclosing module requires `racket/class`, the namespace produced by `namespace-anchor->empty-namespace` certainly contains an instance of `racket/class`. Moreover, that instance is the same as the one imported into the module, so the class datatype is shared.

15.3   Scripting Evaluation and Using `load`

Historically, Lisp implementations did not offer module systems. Instead, large programs were built by essentially scripting the REPL to evaluate program fragments in a particular order. While REPL scripting turns out to be a bad way to structure programs and libraries, it is still sometimes a useful capability.

The `load` function runs a REPL script by `read`ing S-expressions from a file, one by one, and passing them to `eval`. If a file `"place.rkts"` contains

Describing a program via `load` interacts especially badly with macro-defined language extensions [Flatt02].

```
(define city "Salt Lake City")
(define state "Utah")
(printf "~a, ~a\n" city state)
```

then it can be loaded in a REPL:

```
> (load "place.rkts")
Salt Lake City, Utah
> city
"Salt Lake City"
```

Since `load` uses `eval`, however, a module like the following generally will not work—for the same reasons described in §15.1.2 "Namespaces":

```
#lang racket

(define there "Utopia")

(load "here.rkts")
```

The current namespace for evaluating the content of `"here.rkts"` is likely to be empty; in any case, you cannot get `there` from `"here.rkts"`. Also, any definitions in `"here.rkts"` will not become visible for use within the module; after all, the `load` happens dynamically, while references to identifiers within the module are resolved lexically, and therefore statically.

Unlike `eval`, `load` does not accept a namespace argument. To supply a namespace to `load`, set the `current-namespace` parameter. The following example evaluates the expressions in `"here.rkts"` using the bindings of the `racket/base` module:

```racket
#lang racket

(parameterize ([current-namespace (make-base-namespace)])
  (load "here.rkts"))
```

You can even use `namespace-anchor->namespace` to make the bindings of the enclosing module accessible for dynamic evaluation. In the following example, when `"here.rkts"` is `load`ed, it can refer to `there` as well as the bindings of `racket`:

```racket
#lang racket

(define there "Utopia")

(define-namespace-anchor a)
(parameterize ([current-namespace (namespace-anchor->namespace a)])
  (load "here.rkts"))
```

Still, if `"here.rkts"` defines any identifiers, the definitions cannot be directly (i.e., statically) referenced by in the enclosing module.

The `racket/load` module language is different from `racket` or `racket/base`. A module using `racket/load` treats all of its content as dynamic, passing each form in the module body to `eval` (using a namespace that is initialized with `racket`). As a result, uses of `eval` and `load` in the module body see the same dynamic namespace as immediate body forms. For example, if `"here.rkts"` contains

```racket
(define here "Morporkia")
(define (go!) (set! here there))
```

then running

```racket
#lang racket/load

(define there "Utopia")

(load "here.rkts")

(go!)
(printf "~a\n" here)
```

prints "Utopia".

Drawbacks of using `racket/load` include reduced error checking, tool support, and perfor-

mance. For example, with the program

```
#lang racket/load

(define good 5)
(printf "running\n")
good
bad
```

DrRacket's Check Syntax tool cannot tell that the second good is a reference to the first, and the unbound reference to bad is reported only at run time instead of rejected syntactically.

16 Macros

A *macro* is a syntactic form with an associated *transformer* that *expands* the original form into existing forms. To put it another way, a macro is an extension to the Racket compiler. Most of the syntactic forms of `racket/base` and `racket` are actually macros that expand into a small set of core constructs.

Like many languages, Racket provides pattern-based macros that make simple transformations easy to implement and reliable to use. Racket also supports arbitrary macro transformers that are implemented in Racket—or in a macro-extended variant of Racket.

16.1 Pattern-Based Macros

A *pattern-based macro* replaces any code that matches a pattern to an expansion that uses parts of the original syntax that match parts of the pattern.

16.1.1 `define-syntax-rule`

The simplest way to create a macro is to use `define-syntax-rule`:

```
(define-syntax-rule pattern template)
```

As a running example, consider the `swap` macro, which swaps the values stored in two variables. It can be implemented using `define-syntax-rule` as follows:

```
(define-syntax-rule (swap x y)
  (let ([tmp x])
    (set! x y)
    (set! y tmp)))
```

The `define-syntax-rule` form binds a macro that matches a single pattern. The pattern must always start with an open parenthesis followed by an identifier, which is `swap` in this case. After the initial identifier, other identifiers are *macro pattern variables* that can match anything in a use of the macro. Thus, this macro matches the form (swap *form1* *form2*) for any *form_1* and *form_2*.

After the pattern in `define-syntax-rule` is the *template*. The template is used in place of a form that matches the pattern, except that each instance of a pattern variable in the template is replaced with the part of the macro use the pattern variable matched. For example, in

```
(swap first last)
```

The macro is "un-Rackety" in the sense that it involves side effects on variables—but the point of macros is to let you add syntactic forms that some other language designer might not approve.

Macro pattern variables similar to pattern variables for `match`. See §12 "Pattern Matching".

243

the pattern variable x matches `first` and y matches `last`, so that the expansion is

```
(let ([tmp first])
  (set! first last)
  (set! last tmp))
```

16.1.2 Lexical Scope

Suppose that we use the `swap` macro to swap variables named `tmp` and `other`:

```
(let ([tmp 5]
      [other 6])
  (swap tmp other)
  (list tmp other))
```

The result of the above expression should be `(6 5)`. The naive expansion of this use of `swap`, however, is

```
(let ([tmp 5]
      [other 6])
  (let ([tmp tmp])
    (set! tmp other)
    (set! other tmp))
  (list tmp other))
```

whose result is `(5 6)`. The problem is that the naive expansion confuses the `tmp` in the context where `swap` is used with the `tmp` that is in the macro template.

Racket doesn't produce the naive expansion for the above use of `swap`. Instead, it produces

```
(let ([tmp 5]
      [other 6])
  (let ([tmp_1 tmp])
    (set! tmp other)
    (set! other tmp_1))
  (list tmp other))
```

with the correct result in `(6 5)`. Similarly, in the example

```
(let ([set! 5]
      [other 6])
  (swap set! other)
  (list set! other))
```

the expansion is

```
(let ([set!_1 5]
      [other 6])
  (let ([tmp_1 set!_1])
    (set! set!_1 other)
    (set! other tmp_1))
  (list set!_1 other))
```

so that the local `set!` binding doesn't interfere with the assignments introduced by the macro template.

In other words, Racket's pattern-based macros automatically maintain lexical scope, so macro implementors can reason about variable reference in macros and macro uses in the same way as for functions and function calls.

16.1.3 `define-syntax` and `syntax-rules`

The `define-syntax-rule` form binds a macro that matches a single pattern, but Racket's macro system supports transformers that match multiple patterns starting with the same identifier. To write such macros, the programmer much use the more general `define-syntax` form along with the `syntax-rules` transformer form:

```
(define-syntax id
  (syntax-rules (literal-id ...)
    [pattern template]
    ...))
```

For example, suppose we would like a `rotate` macro that generalizes `swap` to work on either two or three identifiers, so that

```
(let ([red 1] [green 2] [blue 3])
  (rotate red green)      ; swaps
  (rotate red green blue) ; rotates left
  (list red green blue))
```

The `define-syntax-rule` form is itself a macro that expands into `define-syntax` with a `syntax-rules` form that contains only one pattern and template.

produces `(1 3 2)`. We can implement `rotate` using `syntax-rules`:

```
(define-syntax rotate
  (syntax-rules ()
    [(rotate a b) (swap a b)]
    [(rotate a b c) (begin
                      (swap a b)
                      (swap b c))]))
```

The expression `(rotate red green)` matches the first pattern in the `syntax-rules` form, so it expands to `(swap red green)`. The expression `(rotate a b c)` matches the second pattern, so it expands to `(begin (swap red green) (swap green blue))`.

16.1.4   Matching Sequences

A better `rotate` macro would allow any number of identifiers, instead of just two or three. To match a use of `rotate` with any number of identifiers, we need a pattern form that has something like a Kleene star. In a Racket macro pattern, a star is written as `...`.

To implement `rotate` with `...`, we need a base case to handle a single identifier, and an inductive case to handle more than one identifier:

```
(define-syntax rotate
  (syntax-rules ()
    [(rotate a) (void)]
    [(rotate a b c ...) (begin
                          (swap a b)
                          (rotate b c ...))]))
```

When a pattern variable like `c` is followed by `...` in a pattern, then it must be followed by `...` in a template, too. The pattern variable effectively matches a sequence of zero or more forms, and it is replaced in the template by the same sequence.

Both versions of `rotate` so far are a bit inefficient, since pairwise swapping keeps moving the value from the first variable into every variable in the sequence until it arrives at the last one. A more efficient `rotate` would move the first value directly to the last variable. We can use `...` patterns to implement the more efficient variant using a helper macro:

```
(define-syntax rotate
  (syntax-rules ()
    [(rotate a c ...)
     (shift-to (c ... a) (a c ...))]))

(define-syntax shift-to
  (syntax-rules ()
    [(shift-to (from0 from ...) (to0 to ...))
     (let ([tmp from0])
       (set! to from) ...
       (set! to0 tmp))]))
```

In the `shift-to` macro, `...` in the template follows `(set! to from)`, which causes the `(set! to from)` expression to be duplicated as many times as necessary to use each identifier matched in the `to` and `from` sequences. (The number of `to` and `from` matches must be the same, otherwise the macro expansion fails with an error.)

16.1.5 Identifier Macros

Given our macro definitions, the `swap` or `rotate` identifiers must be used after an open parenthesis, otherwise a syntax error is reported:

```
> (+ swap 3)
eval:2:0: swap: bad syntax in: swap
```

An *identifier macro* works in any expression. For example, we can define `clock` as an identifier macro that expands to `(get-clock)`, so `(+ clock 3)` would expand to `(+ (get-clock) 3)`. An identifier macro also cooperates with `set!`, and we can define `clock` so that `(set! clock 3)` expands to `(put-clock! 3)`.

The `syntax-id-rules` form is like `syntax-rules`, but it creates a transformer that acts as an identifier macro:

```
(define-syntax id
  (syntax-id-rules (literal-id ...)
    [pattern template]
    ...))
```

Unlike a `syntax-rules` form, the *pattern*s are not required to start with an open parenthesis. Also, `set!` is typically used as a literal to match a use of `set!` in the pattern (as opposed to being a pattern variable.

```
(define-syntax clock
  (syntax-id-rules (set!)
    [(set! clock e) (put-clock! e)]
    [(clock a ...) ((get-clock) a ...)]
    [clock (get-clock)]))
```

The `(clock a ...)` pattern is needed because, when an identifier macro is used after an open parenthesis, the macro transformer is given the whole form, like with a non-identifier macro. Put another way, the `syntax-rules` form is essentially a special case of the `syntax-id-rules` form with errors in the `set!` and lone-identifier cases.

16.1.6 Macro-Generating Macros

Suppose that we have many identifier like `clock` that we'd like to redirect to accessor and mutator functions like `get-clock` and `put-clock!`. We'd like to be able to just write

```
(define-get/put-id clock get-clock put-clock!)
```

Naturally, we can implement `define-get/put-id` as a macro:

```
(define-syntax-rule (define-get/put-id id get put!)
  (define-syntax id
    (syntax-id-rules (set!)
      [(set! id e) (put! e)]
      [(id a (... ...)) ((get) a (... ...))]
      [id (get)])))
```

The `define-get/put-id` macro is a *macro-generating macro*. The only non-obvious part of its definition is the `(... ...)`, which "quotes" `...` so that it takes its usual role in the generated macro, instead of the generating macro.

16.1.7 Extended Example: Call-by-Reference Functions

We can use pattern-matching macros to add a form to Racket for defining first-order *call-by-reference* functions. When a call-by-reference function body mutates its formal argument, the mutation applies to variables that are supplied as actual arguments in a call to the function.

For example, if `define-cbr` is like `define` except that it defines a call-by-reference function, then

```
(define-cbr (f a b)
  (swap a b))

(let ([x 1] [y 2])
  (f x y)
  (list x y))
```

produces `(2 1)`.

We will implement call-by-reference functions by having function calls supply accessor and mutators for the arguments, instead of supplying argument values directly. In particular, for the function `f` above, we'll generate

```
(define (do-f get-a get-b put-a! put-b!)
  (define-get/put-id a get-a put-a!)
  (define-get/put-id b get-b put-b!)
  (swap a b))
```

and redirect a function call `(f x y)` to

```
(do-f (lambda () x)
      (lambda () y)
      (lambda (v) (set! x v))
      (lambda (v) (set! y v)))
```

Clearly, then `define-cbr` is a macro-generating macro, which binds `f` to a macro that expands to a call of `do-f`. That is, `(define-cbr (f a b) (swap ab))` needs to generate the definition

```
(define-syntax f
  (syntax-rules ()
    [(id actual ...)
     (do-f (lambda () actual)
           ...
           (lambda (v)
             (set! actual v))
           ...)]))
```

At the same time, `define-cbr` needs to define `do-f` using the body of `f`, this second part is slightly more complex, so we defer most it to a `define-for-cbr` helper module, which lets us write `define-cbr` easily enough:

```
(define-syntax-rule (define-cbr (id arg ...) body)
  (begin
    (define-syntax id
      (syntax-rules ()
        [(id actual (... ...))
         (do-f (lambda () actual)
               (... ...)
               (lambda (v)
                 (set! actual v))
               (... ...))]))
    (define-for-cbr do-f (arg ...)
      () ; explained below...
      body)))
```

Our remaining task is to define `define-for-cbr` so that it converts

```
(define-for-cbr do-f (a b) () (swap a b))
```

to the function definition `do-f` above. Most of the work is generating a `define-get/put-id` declaration for each argument, `a` ad `b`, and putting them before the body. Normally, that's an easy task for `...` in a pattern and template, but this time there's a catch: we need to generate the names `get-a` and `put-a!` as well as `get-b` and `put-b!`, and the pattern language provides no way to synthesize identifiers based on existing identifiers.

As it turns out, lexical scope gives us a way around this problem. The trick is to iterate expansions of `define-for-cbr` once for each argument in the function, and that's why `define-cbr` starts with an apparently useless `()` after the argument list. We need to keep track of all the arguments seen so far and the `get` and `put` names generated for each, in addition to the arguments left to process. After we've processed all the identifiers, then we have all the names we need.

Here is the definition of `define-for-cbr`:

```
(define-syntax define-for-cbr
  (syntax-rules ()
    [(define-for-cbr do-f (id0 id ...)
       (gens ...) body)
     (define-for-cbr do-f (id ...)
       (gens ... (id0 get put)) body)]
    [(define-for-cbr do-f ()
       ((id get put) ...) body)
     (define (do-f get ... put ...)
       (define-get/put-id id get put) ...
       body)]))
```

Step-by-step, expansion proceeds as follows:

```
(define-for-cbr do-f (a b)
  () (swap a b))
=> (define-for-cbr do-f (b)
     ([a get_1 put_1]) (swap a b))
=> (define-for-cbr do-f ()
     ([a get_1 put_1] [b get_2 put_2]) (swap a b))
=> (define (do-f get_1 get_2 put_1 put_2)
     (define-get/put-id a get_1 put_1)
     (define-get/put-id b get_2 put_2)
     (swap a b))
```

The "subscripts" on `get_1`, `get_2`, `put_1`, and `put_2` are inserted by the macro expander to preserve lexical scope, since the `get` generated by each iteration of `define-for-cbr` should not bind the `get` generated by a different iteration. In other words, we are essentially tricking the macro expander into generating fresh names for us, but the technique illustrates some of the surprising power of pattern-based macros with automatic lexical scope.

The last expression eventually expands to just

```
(define (do-f get_1 get_2 put_1 put_2)
  (let ([tmp (get_1)])
    (put_1 (get_2))
    (put_2 tmp)))
```

which implements the call-by-name function `f`.

To summarize, then, we can add call-by-reference functions to Racket with just three small pattern-based macros: `define-cbr`, `define-for-cbr`, and `define-get/put-id`.

16.2　General Macro Transformers

The `define-syntax` form creates a *transformer binding* for an identifier, which is a binding that can be used at compile time while expanding expressions to be evaluated at run time. The compile-time value associated with a transformer binding can be anything; if it is a procedure of one argument, then the binding is used as a macro, and the procedure is the *macro transformer*.

The `syntax-rules` and `syntax-id-rules` forms are macros that expand to procedure forms. For example, if you evaluate a `syntax-rules` form directly (instead of placing on the right-hand of a `define-syntax` form), the result is a procedure:

```
> (syntax-rules () [(nothing) something])
#<procedure>
```

Instead of using `syntax-rules`, you can write your own macro transformer procedure directly using `lambda`. The argument to the procedure is a values that represents the source form, and the result of the procedure must be a value that represents the replacement form.

16.2.1　Syntax Objects

The input and output of a macro transformer (i.e., source and replacement forms) are represented as *syntax objects*. A syntax object contains symbols, lists, and constant values (such as numbers) that essentially correspond to the `quoted` form of the expression. For example, a representation of the expression `(+ 1 2)` contains the symbol `'+` and the numbers `1` and `2`, all in a list. In addition to this quoted content, a syntax object associates source-location and lexical-binding information with each part of the form. The source-location information is used when reporting syntax errors (for example), and the lexical-biding information allows the macro system to maintain lexical scope. To accommodate this extra information, the represention of the expression `(+ 1 2)` is not merely `'(+ 1 2)`, but a packaging of `'(+ 1 2)` into a syntax object.

To create a literal syntax object, use the `syntax` form:

```
> (syntax (+ 1 2))
#<syntax:1:0 (+ 1 2)>
```

In the same way that `'` abbreviates `quote`, `#'` abbreviates `syntax`:

```
> #'(+ 1 2)
#<syntax:1:0 (+ 1 2)>
```

A syntax object that contains just a symbol is an *identifier syntax object*. Racket provides some additional operations specific to identifier syntax objects, including the `identifier?` operation to detect identifiers. Most notably, `free-identifier=?` determines whether two

identifiers refer to the same binding:

```
> (identifier? #'car)
#t
> (identifier? #'(+ 1 2))
#f
> (free-identifier=? #'car #'cdr)
#f
> (free-identifier=? #'car #'car)
#t
> (require (only-in racket/base [car also-car]))
> (free-identifier=? #'car #'also-car)
#t
> (free-identifier=? #'car (let ([car 8])
                            #'car))
#f
```

The last example above, in particular, illustrates how syntax objects preserve lexical-context information.

To see the lists, symbols, numbers, etc. within a syntax object, use `syntax->datum`:

```
> (syntax->datum #'(+ 1 2))
'(+ 1 2)
```

The `syntax-e` function is similar to `syntax->datum`, but it unwraps a single layer of source-location and lexical-context information, leaving sub-forms that have their own information wrapped as syntax objects:

```
> (syntax-e #'(+ 1 2))
'(#<syntax:1:0 +> #<syntax:1:0 1> #<syntax:1:0 2>)
```

The `syntax-e` function always leaves syntax-object wrappers around sub-forms that are represented via symbols, numbers, and other literal values. The only time it unwraps extra sub-forms is when unwrapping a pair, in which case the `cdr` of the pair may be recursively unwrapped, depending on how the syntax object was constructed.

The opposite of `syntax->datum` is, of course, `datum->syntax`. In addition to a datum like `'(+ 1 2)`, `datum->syntax` needs an existing syntax object to donate its lexical context, and optionally another syntax object to donate its source location:

```
> (datum->syntax #'lex
                 '(+ 1 2)
                 #'srcloc)
#<syntax:1:0 (+ 1 2)>
```

In the above example, the lexical context of #'lex is used for the new syntax object, while

the source location of #'`srcloc` is used.

When the second (i.e., the "datum") argument to `datum->syntax` includes syntax objects, those syntax objects are preserved intact in the result. That is, deconstructing the result with `syntax-e` eventually produces the syntax objects that were given to `datum->syntax`.

16.2.2 Mixing Patterns and Expressions: `syntax-case`

The procedure generated by `syntax-rules` internally uses `syntax-e` to deconstruct the given syntax object, and it uses `datum->syntax` to construct the result. The `syntax-rules` form doesn't provide a way to escape from pattern-matching and template-construction mode into an arbitrary Racket expression.

The `syntax-case` form lets you mix pattern matching, template construction, and arbitrary expressions:

```
(syntax-case stx-expr (literal-id ...)
  [pattern expr]
  ...)
```

Unlike `syntax-rules`, the `syntax-case` form does not produce a procedure. Instead, it starts with a *stx-expr* expression that determines the syntax object to match against the *pattern*s. Also, each `syntax-case` clause has a *pattern* and *expr*, instead of a *pattern* and *template*. Within an *expr*, the syntax form—usually abbreviated with `#'`—shifts into template-construction mode; if the *expr* of a clause starts with `#'`, then we have something like a `syntax-rules` form:

```
> (syntax->datum
   (syntax-case #'(+ 1 2) ()
     [(op n1 n2) #'(- n1 n2)]))
'(- 1 2)
```

We could write the `swap` macro using `syntax-case` instead of `define-syntax-rule` or `syntax-rules`:

```
(define-syntax swap
  (lambda (stx)
    (syntax-case stx ()
      [(swap x y) #'(let ([tmp x])
                      (set! x y)
                      (set! y tmp))])))
```

One advantage of using `syntax-case` is that we can provide better error reporting for `swap`. For example, with the `define-syntax-rule` definition of `swap`, then (`swap x 2`) pro-

duces a syntax error in terms of `set!`, because `2` is not an identifier. We can refine our `syntax-case` implementation of `swap` to explicitly check the sub-forms:

```
(define-syntax swap
  (lambda (stx)
    (syntax-case stx ()
      [(swap x y)
       (if (and (identifier? #'x)
                (identifier? #'y))
           #'(let ([tmp x])
               (set! x y)
               (set! y tmp))
           (raise-syntax-error #f
                               "not an identifier"
                               stx
                               (if (identifier? #'x)
                                   #'y
                                   #'x)))])))
```

With this definition, `(swap x 2)` provides a syntax error originating from `swap` instead of `set!`.

In the above definition of `swap`, `#'x` and `#'y` are templates, even though they are not used as the result of the macro transformer. This example illustrates how templates can be used to access pieces of the input syntax, in this case for checking the form of the pieces. Also, the match for `#'x` or `#'y` is used in the call to `raise-syntax-error`, so that the syntax-error message can point directly to the source location of the non-identifier.

16.2.3 `with-syntax` **and** `generate-temporaries`

Since `syntax-case` lets us compute with arbitrary Racket expression, we can more simply solve a problem that we had in writing `define-for-cbr` (see §16.1.7 "Extended Example: Call-by-Reference Functions"), where we needed to generate a set of names based on a sequence `id ...`:

```
(define-syntax (define-for-cbr stx)
  (syntax-case stx ()
    [(_ do-f (id ...) body)
     ....
       #'(define (do-f get ... put ...)
           (define-get/put-id id get put) ...
           body) ....]))
```

In place of the `....`s above, we need to bind `get ...` and `put ...` to lists of generated identifiers. We cannot use `let` to bind `get` and `put`, because we need bindings that count

This example uses `(define-syntax (id arg) body ...+)`, which is equivalent to `(define-syntax id (lambda (arg) body ...+))`.

254

as pattern variables, instead of normal local variables. The `with-syntax` form lets us bind pattern variables:

```
(define-syntax (define-for-cbr stx)
  (syntax-case stx ()
    [(_ do-f (id ...) body)
     (with-syntax ([(get ...) ....]
                   [(put ...) ....])
       #'(define (do-f get ... put ...)
           (define-get/put-id id get put) ...
           body))]))
```

Now we need an expression in place of .... that generates as many identifiers as there are `id` matches in the original pattern. Since this is a common task, Racket provides a helper function, `generate-temporaries`, that takes a sequece of identifiers and returns a sequence of generated identifiers:

```
(define-syntax (define-for-cbr stx)
  (syntax-case stx ()
    [(_ do-f (id ...) body)
     (with-syntax ([(get ...) (generate-temporaries #'(id ...))]
                   [(put ...) (generate-temporaries #'(id ...))])
       #'(define (do-f get ... put ...)
           (define-get/put-id id get put) ...
           body))]))
```

This way of generating identifiers is normally easier to think about than tricking the macro expander into generating names with purely pattern-based macros.

In general, the right-hand side of a `with-syntax` binding is a pattern, just like in `syntax-case`. In fact, a `with-syntax` form is just a `syntax-case` form turned partially inside-out.

16.2.4 Compile and Run-Time Phases

As sets of macros get more complicated, you might want to write your own helper functions, like `generate-temporaries`. For example, to provide good syntax-error messsage, `swap`, `rotate`, and `define-cbr` all should check that certain sub-forms in the source form are identifiers. We could use a `check-ids` to perform this checking everywhere:

```
(define-syntax (swap stx)
  (syntax-case stx ()
    [(swap x y) (begin
                  (check-ids stx #'(x y))
                  #'(let ([tmp x])
                      (set! x y)
```

255

```
                         (set! y tmp)))])))

  (define-syntax (rotate stx)
    (syntax-case stx ()
      [(rotate a c ...)
       (begin
         (check-ids stx #'(a c ...))
         #'(shift-to (c ... a) (a c ...)))]))
```

The `check-ids` function can use the `syntax->list` function to convert a syntax-object wrapping a list into a list of syntax objects:

```
  (define (check-ids stx forms)
    (for-each
     (lambda (form)
       (unless (identifier? form)
         (raise-syntax-error #f
                             "not an identifier"
                             stx
                             form)))
     (syntax->list forms)))
```

If you define `swap` and `check-ids` in this way, however, it doesn't work:

```
  > (let ([a 1] [b 2]) (swap a b))
  reference to undefined identifier: check-ids
```

The problem is that `check-ids` is defined as a run-time expression, but `swap` is trying to use it at compile time. In interactive mode, compile time and run time are interleaved, but they are not interleaved within the body of a module, and they are not interleaved or across modules that are compiled ahead-of-time. To help make all of these modes treat code consistently, Racket separates the binding spaces for different phases.

To define a `check-ids` function that can be referenced at compile time, use `define-for-syntax`:

```
  (define-for-syntax (check-ids stx forms)
    (for-each
     (lambda (form)
       (unless (identifier? form)
         (raise-syntax-error #f
                             "not an identifier"
                             stx
                             form)))
     (syntax->list forms)))
```

With this for-syntax definition, then `swap` works:

```
> (let ([a 1] [b 2]) (swap a b) (list a b))
'(2 1)
> (swap a 1)
eval:7:0: swap: not an identifier at: 1 in: (swap a 1)
```

When organizing a program into modules, you may want to put helper functions in one module to be used by macros that reside on other modules. In that case, you can write the helper function using define:

"utils.ss"

```
#lang racket

(provide check-ids)

(define (check-ids stx forms)
  (for-each
   (lambda (form)
     (unless (identifier? form)
       (raise-syntax-error #f
                           "not an identifier"
                           stx
                           form)))
   (syntax->list forms)))
```

Then, in the module that implements macros, import the helper function using (require (for-syntax "utils.ss")) instead of (require "utils.ss"):

```
#lang racket

(require (for-syntax "utils.ss"))

(define-syntax (swap stx)
  (syntax-case stx ()
    [(swap x y) (begin
                  (check-ids stx #'(x y))
                  #'(let ([tmp x])
                      (set! x y)
                      (set! y tmp)))]))
```

Since modules are separately compiled and cannot have circular dependencies, the "utils.ss" module's run-time body can be compiled before the compiling the module that implements swap. Thus, the run-time definitions in "utils.ss" can be used to implement swap, as long as they are explicitly shifted into compile time by (require (for-syntax ....)).

The racket module provides syntax-case, generate-temporaries, lambda, if, and more for use in both the run-time and compile-time phases. That is why we can use syntax-

case in the `racket` REPL both directly and in the right-hand side of a `define-syntax` form.

The `racket/base` module, in contrast, exports those bindings only in the run-time phase. If you change the module above that defines `swap` so that it uses the `racket/base` language instead of `racket`, then it no longer works. Adding `(require (for-syntax racket/base))` imports `syntax-case` and more into the compile-time phase, so that the module works again.

Suppose that `define-syntax` is used to define a local macro in the right-hand side of a `define-syntax` form. In that case, the right-hand side of the inner `define-syntax` is in the *meta-compile phase level*, also known as *phase level 2*. To import `syntax-case` into that phase level, you would have to use `(require (for-syntax (for-syntax racket/base)))` or, equivalently, `(require (for-meta 2 racket/base))`.

Negative phase levels also exist. If a macro uses a helper function that is imported `for-syntax`, and if the helper function returns syntax-object constants generated by `syntax`, then identifiers in the syntax will need bindings at *phase level -1*, also known as the *template phase level*, to have any binding at the run-time phase level relative to the module that defines the macro.

16.2.5 Syntax Certificates

A use of a macro can expand into a use of an identifier that is not exported from the module that binds the macro. In general, such an identifier must not be extracted from the expanded expression and used in a different context, because using the identifier in a different context may break invariants of the macro's module.

For example, the following module exports a macro `go` that expands to a use of `unchecked-go`:

```
(module m mzscheme
  (provide go)
  (define (unchecked-go n x)
    ; to avoid disaster, n must be a number
    (+ n 17))
  (define-syntax (go stx)
    (syntax-case stx ()
      [(_ x)
       #'(unchecked-go 8 x)])))
```

If the reference to `unchecked-go` is extracted from the expansion of `(go 'a)`, then it might be inserted into a new expression, `(unchecked-go #f 'a)`, leading to disaster. The `datum->syntax` procedure can be used similarly to construct references to an unexported identifier, even when no macro expansion includes a reference to the identifier.

To prevent such abuses of unexported identifiers, the expander rejects references to unexported identifiers unless they appear in *certified* syntax objects. The macro expander always certifies a syntax object that is produced by a transformer. For example, when `(go 'a)` is expanded to `(unchecked-go 8 'a)`, a certificate is attached to the result `(unchecked-go 8 'a)`. Extracting just `unchecked-go` removes the identifier from the certified expression, so that the reference is disallowed when it is inserted into `(unchecked-go #f 'a)`. The `expand` and `local-expand` (when used with an empty stop list) functions lift all certificates to the outermost result expression, except as indicated by `'certify-mode` syntax properties (see §11.8 "Syntax Certificates").

In addition to checking module references, the macro expander disallows references to local bindings where the binding identifier is less certified than the reference. Otherwise, the expansion of `(go 'a)` could be wrapped with a local binding that redirects #%app to `values`, thus obtaining the value of `unchecked-go`. Note that a capturing #%app would have to be extracted from the expansion of `(go 'a)`, since lexical scope would prevent an arbitrary #%app from capturing. The act of extracting #%app removes its certification, whereas the #%app within the expansion is still certified; comparing these certifications, the macro expander rejects the local-binding reference, and `unchecked-go` remains protected.

In much the same way that the macro expander copies properties from a syntax transformer's input to its output (see §11.7 "Syntax Object Properties"), the expander copies certificates from a transformer's input to its output. Building on the previous example,

```
(module n mzscheme
  (require m)
  (provide go-more)
  (define y 'hello)
  (define-syntax (go-more stx)
    #'(go y)))
```

the expansion of `(go-more)` introduces a reference to the unexported `y` in `(go y)`, and a certificate allows the reference to `y`. As `(go y)` is expanded to `(unchecked-go 8 y)`, the certificate that allows `y` is copied over, in addition to the certificate that allows the reference to `unchecked-go`.

When a protected identifier becomes inaccessible by direct reference (i.e., when the current code inspector is changed so that it does not control the module's invocation; see §13.10 "Code Inspectors"), the protected identifier is treated like an unexported identifier.

Certificate Propagation

When the result of a macro expansion contains a `quote-syntax` form, the macro expansion's certificate must be attached to the resulting syntax object to support macro-generating macros. In general, when the macro expander encounters `quote-syntax`, it attaches all certificates from enclosing expressions to the quoted syntax constant. However, the certifi-

cates are attached to the syntax constant as *inactive* certificates, and inactive certificates do not count directly for certifying identifier access. Inactive certificates become active when the macro expander certifies the result of a macro expansion; at that time, the expander removes all inactive certificates within the expansion result and attaches active versions of the certificates to the overall expansion result.

For example, suppose that the go macro is implemented through a macro:

```
(module m mzscheme
  (provide def-go)
  (define (unchecked-go n x)
    (+ n 17))
  (define-syntax (def-go stx)
   (syntax-case stx ()
     [(_ go)
      #'(define-syntax (go stx)
          (syntax-case stx ()
            [(_ x)
             #'(unchecked-go 8 x)]))])))
```

When def-go is used inside another module, the generated macro should legally generate expressions that use unchecked-go, since def-go in m had complete control over the generated macro.

```
(module n mzscheme
  (require m)
  (def-go go)
  (go 10))
```

This example works because the expansion of (def-go go) is certified to access protected identifiers in m, including unchecked-go. Specifically, the certified expansion is a definition of the macro go, which includes a syntax-object constant unchecked-go. Since the enclosing macro declaration is certified, the unchecked-go syntax constant gets an inactive certificate to access protected identifiers of m. When (go 10) is expanded, the inactive certificate on unchecked-go is activated for the macro result (unchecked-go 8 10), and the access of unchecked-go is allowed.

To see why unchecked-go as a syntax constant must be given an inactive certificate instead of an active one, it's helpful to write the def-go macro as follows:

```
(define-syntax (def-go stx)
 (syntax-case stx ()
   [(_ go)
    #'(define-syntax (go stx)
        (syntax-case stx ()
          [(_ x)
           (with-syntax ([ug (quote-syntax unchecked-go)])
```

260

```
        #'(ug 8 x))])))])))
```

In this case, `unchecked-go` is clearly quoted as an immediate syntax object in the expansion of (`def-go go`). If this syntax object were given an active certificate, then it would keep the certificate—directly on the identifier `unchecked-go`—in the result (`unchecked-go 8 10`). Consequently, the `unchecked-go` identifier could be extracted and used with its certificate intact. Attaching an inactive certificate to `unchecked-go` and activating it only for the complete result (`unchecked-go 8 10`) ensures that `unchecked-go` is used only in the way intended by the implementor of `def-go`.

The `datum->syntax` procedure allows inactive certificates to be transferred from one syntax object to another. Such transfers are allowed because a macro transformer with access to the syntax object could already wrap it with an arbitrary context before activating the certificates. In practice, transferring inactive certificates is useful mainly to macros that implement new template forms, such as `syntax/loc`.

Internal Certificates

In some cases, a macro implementor intends to allow limited destructuring of a macro result without losing the result's certificate. For example, given the following `define-like-y` macro,

```
(module q mzscheme
  (provide define-like-y)
  (define y 'hello)
  (define-syntax (define-like-y stx)
    (syntax-case stx ()
      [(_ id) #'(define-values (id) y)])))
```

someone may use the macro in an internal definition:

```
(let ()
  (define-like-y x)
  x)
```

The implementor of the `q` module most likely intended to allow such uses of `define-like-y`. To convert an internal definition into a `letrec` binding, however, the `define` form produced by `define-like-y` must be deconstructed, which would normally lose the certificate that allows the reference to `y`.

The internal use of `define-like-y` is allowed because the macro expander treats specially a transformer result that is a syntax list beginning with `define-values`. In that case, instead of attaching the certificate to the overall expression, the certificate is instead attached to each individual element of the syntax list, pushing the certificates into the second element of the list so that they are attached to the defined identifiers. Thus, a certificate is attached

261

to `define-values`, `x`, and `y` in the expansion result `(define-values (x) y)`, and the definition can be deconstructed for conversion to `letrec`.

Just like the new certificate that is added to a transformer result, old certificates from the input are similarly moved to syntax-list elements when the result starts with `define-values`. Thus, `define-like-y` could have been implemented to produce `(define id y)`, using `define` instead of `define-values`. In that case, the certificate to allow reference to `y` would be attached initially to the expansion result `(define x y)`, but as the `define` is expanded to `define-values`, the certificate would be moved to the parts.

The macro expander treats syntax-list results starting with `define-syntaxes` in the same way that it treats results starting with `define-values`. Syntax-list results starting with `begin` are treated similarly, except that the second element of the syntax list is treated like all the other elements (i.e., the certificate is attached to the element instead of its content). Furthermore, the macro expander applies this special handling recursively, in case a macro produces a `begin` form that contains nested `define-values` forms.

The default application of certificates can be overridden by attaching a `'certify-mode` property (see §11.7 "Syntax Object Properties") to the result syntax object of a macro transformer. If the property value is `'opaque`, then the certificate is attached to the syntax object and not its parts. If the property value is `'transparent`, then the certificate is attached to the syntax object's parts. If the property value is `'transparent-binding`, then the certificate is attached to the syntax object's parts and to the sub-parts of the second part (as for `define-values` and `define-syntaxes`). The `'transparent` and `'transparent-binding` modes triggers recursive property checking at the parts, so that the certificate can be pushed arbitrarily deep into a transformer's result.

17 Creating Languages

The macro facilities defined in the preceding chapter let a programmer define syntactic extensions to a language, but a macro is limited in two ways:

- a macro cannot restrict the syntax available in its context or change the meaning of surrounding forms; and

- a macro can extend the syntax of a language only within the parameters of the language's lexical conventions, such as using parentheses to group the macro name with its subforms and using the core syntax of identifiers, keywords, and literals.

That is, a macro can only extend a language, and it can do so only at the expander layer. Racket offers additional facilities for defining a starting point of the expander layer, for extending the reader layer, for defining the starting point of the reader layer, and for packaging a reader and expander starting point into a conveniently named language.

17.1 Module Languages

When using the longhand `module` form for writing modules, the module path that is specified after the new module's name provides the initial imports for the module. Since the initial-import module determines even the most basic bindings that are available in a module's body, such as `require`, the initial import can be called a *module language*.

The most common module languages are `racket` or `racket/base`, but you can define your own module language by defining a suitable module. For example, using `provide` subforms like `all-from-out`, `except-out`, and `rename-out`, you can add, remove, or rename bindings from `racket` to produce a module language that is a variant of `racket`:

```
> (module raquet racket
    (provide (except-out (all-from-out racket) lambda)
             (rename-out [lambda function])))
> (module score 'raquet
    (map (function (points) (case points
                              [(0) "love"] [(1) "fifteen"]
                              [(2) "thirty"] [(3) "forty"]))
         (list 0 2)))
> (require 'score)
'("love" "thirty")
```

17.1.1 Implicit Form Bindings

If you try to remove too much from `racket` in defining your own module language, then the resulting module will no longer work right as a module language:

```
> (module just-lambda racket
    (provide lambda))
> (module identity 'just-lambda
    (lambda (x) x))
eval:2:0: module: no #%module-begin binding in the module's
language in: (module identity (quote just-lambda) (lambda
(x) x))
```

The `#%module-begin` form is an implicit form that wraps the body of a module. It must be provided by a module that is to be used as module language:

```
> (module just-lambda racket
    (provide lambda #%module-begin))
> (module identity 'just-lambda
    (lambda (x) x))
> (require 'identity)
#<procedure>
```

The other implicit forms provided by `racket/base` are `#%app` for function calls, `#%datum` for literals, and `#%top` for identifiers that have no binding:

```
> (module just-lambda racket
    (provide lambda #%module-begin
             ; ten needs these, too:
             #%app #%datum))
> (module ten 'just-lambda
    ((lambda (x) x) 10))
> (require 'ten)
10
```

Implicit forms such as `#%app` can be used explicitly in a module, but they exist mainly to allow a module language to restrict or change the meaning of implicit uses. For example, a `lambda-calculus` module language might restrict functions to a single argument, restrict function calls to supply a single argument, restrict the module body to a single expression, disallow literals, and treat unbound identifiers as uninterpreted symbols:

```
> (module lambda-calculus racket
    (provide (rename-out [1-arg-lambda lambda]
                         [1-arg-app #%app]
                         [1-form-module-begin #%module-begin]
                         [no-literals #%datum]
```

```
                              [unbound-as-quoted #%top]))
      (define-syntax-rule (1-arg-lambda (x) expr)
        (lambda (x) expr))
      (define-syntax-rule (1-arg-app e1 e2)
        (#%app e1 e2))
      (define-syntax-rule (1-form-module-begin e)
        (#%module-begin e))
      (define-syntax (no-literals stx)
        (raise-syntax-error #f "no" stx))
      (define-syntax-rule (unbound-as-quoted . id)
        'id))
> (module ok 'lambda-calculus
    ((lambda (x) (x z))
     (lambda (y) y)))
> (require 'ok)
'z
> (module not-ok 'lambda-calculus
    (lambda (x y) x))
eval:4:0: lambda: (lambda (x y) x) did not match pattern
(lambda (x) expr) in: (lambda (x y) x)
> (module not-ok 'lambda-calculus
    (lambda (x) x)
    (lambda (y) (y y)))
eval:5:0: #%module-begin: (#%module-begin (lambda (x) x)
(lambda (y) (y y))) did not match pattern (#%module-begin
e) in: (#%module-begin (lambda (x) x) (lambda (y) (y y)))
> (module not-ok 'lambda-calculus
    (lambda (x) (x x x)))
eval:6:0: #%app: (#%app x x x) did not match pattern (#%app
e1 e2) in: (#%app x x x)
> (module not-ok 'lambda-calculus
    10)
eval:7:0: #%datum: no in: (#%datum . 10)
```

Module languages rarely redefine #%app, #%datum, and #%top, but redefining #%module-begin is more frequently useful. For example, when using modules to construct descriptions of HTML pages where a description is exported from the module as page, an alternate #%module-begin can help eliminate provide and quasiquoting boilerplate, as in "html.rkt":

"html.rkt"

```
#lang racket
(require racket/date)

(provide (except-out (all-from-out racket)
                     #%module-begin)
```

```
           (rename-out [module-begin #%module-begin])
           now)

  (define-syntax-rule (module-begin expr ...)
    (#%module-begin
     (define page '(html expr ...))
     (provide page)))

  (define (now)
    (parameterize ([date-display-format 'iso-8601])
      (date->string (seconds->date (current-seconds)))))
```

Using the "html.rkt" module language, a simple web page can be described without having to explicitly define or export page and starting in quasiquoted mode instead of expression mode:

```
  > (module lady-with-the-spinning-head "html.rkt"
      (title "Queen of Diamonds")
      (p "Updated: " ,(now)))
  > (require 'lady-with-the-spinning-head)
  > page
  '(html (title "Queen of Diamonds") (p "Updated: " "2011-02-14"))
```

17.1.2   Using #lang s-exp

Implementing a language at the level of #lang is more complex than declaring a single module, because #lang lets programmers control several different facets of a language. The s-exp language, however, acts as a kind of meta-language for using a module language with the #lang shorthand:

```
  #lang s-exp module-name
  form ...
```

is the same as

```
  (module name module-name
    form ...)
```

where name is derived from the source file containing the #lang program. The name s-exp is short for "S-expression," which is a traditional name for Racket's reader-level lexical conventions: parentheses, identifiers, numbers, double-quoted strings with certain backslash escapes, and so on.

Using #lang s-exp, the lady-with-the-spinning-head example from before can be written more compactly as:

```
#lang s-exp "html.rkt"

(title "Queen of Diamonds")
(p "Updated: " ,(now))
```

Later in this guide, §17.3 "Defining new #lang Languages" explains how to define your own #lang language, but first we explain how you can write reader-level extensions to Racket.

17.2   Reader Extensions

The reader layer of the Racket language can be extended through the #reader form. A reader extension is implemented as a module that is named after #reader. The module exports functions that parse raw characters into a form to be consumed by the expander layer.

The syntax of #reader is

> #reader ⟨*module-path*⟩ ⟨*reader-specific*⟩

where ⟨*module-path*⟩ names a module that provides read and read-syntax functions. The ⟨*reader-specific*⟩ part is a sequence of characters that is parsed as determined by the read and read-syntax functions from ⟨*module-path*⟩.

For example, suppose that file "five.rkt" contains

<div align="right">

"five.rkt"

</div>

```
#lang racket/base

(provide read read-syntax)

(define (read in) (list (read-string 5 in)))
(define (read-syntax src in) (list (read-string 5 in)))
```

Then, the program

```
#lang racket/base

'(1 #reader"five.rkt"234567 8)
```

is equivalent to

```
#lang racket/base

'(1 ("23456") 7 8)
```

because the read and read-syntax functions of "five.rkt" both read five characters

from the input stream and put them into a string and then a list. The reader functions from "five.rkt" are not obliged to follow Racket lexical conventions and treat the continuous sequence 234567 as a single number. Since only the 23456 part is consumed by `read` or `read-syntax`, the 7 remains to be parsed in the usual Racket way. Similarly, the reader functions from "five.rkt" are not obliged to ignore whitespace, and

```
#lang racket/base

'(1 #reader"five.rkt" 234567 8)
```

is equivalent to

```
#lang racket/base

'(1 (" 2345") 67 8)
```

since the first character immediately after "five.rkt" is a space.

A `#reader` form can be used in the REPL, too:

```
> '#reader"five.rkt"abcde
'("abcde")
```

17.2.1   Source Locations

The difference between `read` and `read-syntax` is that `read` is meant to be used for data while `read-syntax` is meant to be used to parse programs. More precisely, the `read` function will be used when the enclosing stream is being parsed by the Racket `read`, and `read-syntax` is used when the enclosing stream is being parsed by the Racket `read-syntax` function. Nothing requires `read` and `read-syntax` to parse input in the same way, but making them different would confuse programmers and tools.

The `read-syntax` function can return the same kind of value as `read`, but it should normally return a syntax object that connects the parsed expression with source locations. Unlike the "five.rkt" example, the `read-syntax` function is typically implemented directly to produce syntax objects, and then `read` can use `read-syntax` and strip away syntax object wrappers to produce a raw result.

The following "arith.rkt" module implements a reader to parse simple infix arithmetic expressions into Racket forms. For example, 1*2+3 parses into the Racket form `(+ (* 1 2) 3)`. The supported operators are +, -, *, and /, while operands can be unsigned integers or single-letter variables. The implementation uses `port-next-location` to obtain the current source location, and it uses `datum->syntax` to turn raw values into syntax objects.

"arith.rkt"

```
#lang racket
```

268

```
(require syntax/readerr)

(provide read read-syntax)

(define (read in)
  (syntax->datum (read-syntax #f in)))

(define (read-syntax src in)
  (skip-whitespace in)
  (read-arith src in))

(define (skip-whitespace in)
  (regexp-match #px"^\\s*" in))

(define (read-arith src in)
  (define-values (line col pos) (port-next-location in))
  (define expr-match
    (regexp-match
     ; Match an operand followed by any number of
     ; operator-operand sequences, and prohibit an
     ; additional operator from following immediately:
     #px"^([a-z]|[0-9]+)(?:[-+*/]([a-z]|[0-9]+))*(?![-+*/])"
     in))

  (define (to-syntax v delta span-str)
    (datum->syntax #f v (make-srcloc delta span-str)))
  (define (make-srcloc delta span-str)
    (and line
         (vector src line (+ col delta) (+ pos delta)
                 (string-length span-str))))

  (define (parse-expr s delta)
    (match (or (regexp-match #rx"^(.*?)([+-])(.*)$" s)
               (regexp-match #rx"^(.*?)([*/])(.*)$" s))
      [(list _ a-str op-str b-str)
       (define a-len (string-length a-str))
       (define a (parse-expr a-str delta))
       (define b (parse-expr b-str (+ delta 1 a-len)))
       (define op (to-syntax (string->symbol op-str)
                             (+ delta a-len) op-str))
       (to-syntax (list op a b) delta s)]
      [else (to-syntax (or (string->number s)
                           (string->symbol s))
                       delta s)]))

  (unless expr-match
```

```
      (raise-read-error "bad arithmetic syntax"
                        src line col pos
                        (and pos (- (file-position in) pos)))))
  (parse-expr (bytes->string/utf-8 (car expr-match)) 0))
```

If the `"arith.rkt"` reader is used in an expression position, then its parse result will be treated as a Racket expression. If it is used in a quoted form, however, then it just produces a number or a list:

```
> #reader"arith.rkt" 1*2+3
5
> '#reader"arith.rkt" 1*2+3
'(+ (* 1 2) 3)
```

The `"arith.rkt"` reader could also be used in positions that make no sense. Since the `read-syntax` implementation tracks source locations, syntax errors can at least refer to parts of the input in terms of their original locations (at the beginning of the error message):

```
> (let #reader"arith.rkt" 1*2+3 8)
repl:1:27: let: bad syntax (not an identifier and
expression for a binding) at: + in: (let (+ (* 1 2) 3) 8)
```

17.2.2   Readtables

A reader extension's ability to parse input characters in an arbitrary way can be powerful, but many cases of lexical extension call for a less general but more composable approach. In much the same way that the expander level of Racket syntax can be extended through macros, the reader level of Racket syntax can be composably extended through a *readtable*.

The Racket reader is a recursive-descent parser, and the readtable maps characters to parsing handlers. For example, the default readtable maps `(` to a handler that recursively parses subforms until it finds a `)`. The `current-readtable` parameter determines the readtable that is used by `read` or `read-syntax`. Rather than parsing raw characters directly, a reader extension can install an extended readtable and then chain to `read` or `read-syntax`.

See §4.13 "Dynamic Binding: `parameterize`" for an introduction to parameters.

The `make-readtable` function constructs a new readtable as an extension of an existing one. It accepts a sequence of specifications in terms of a character, a type of mapping for the character, and (for certain types of mappings) a parsing procedure. For example, to extend the readtable so that `$` can be used to start and end infix expressions, implement a `parse-dollar` function and use:

```
(make-readtable (current-readtable)
                #\$ 'terminating-macro read-dollar)
```

The protocol for `read-dollar` requires the function to accept different numbers of argu-

ments depending on whether it is being used in read or read-syntax mode. In read mode, the parser function is given two arguments: the character that triggered the parser function and the input port that is being read. In read-syntax mode, the function must accept four additional arguments that provide the source location of the character.

The following "dollar.rkt" module defines a parse-dollar function in terms of the read and read-syntax functions provided by "arith.rkt", and it puts parse-dollar together with new read and read-syntax functions that install the readtable and chain to Racket's read or read-syntax:

<div align="right">

"dollar.rkt"

</div>

```racket
#lang racket
(require syntax/readerr
         (prefix-in arith: "arith.rkt"))

(provide (rename-out [$-read read]
                     [$-read-syntax read-syntax]))

(define ($-read in)
  (parameterize ([current-readtable (make-$-readtable)])
    (read in)))

(define ($-read-syntax src in)
  (parameterize ([current-readtable (make-$-readtable)])
    (read-syntax src in)))

(define (make-$-readtable)
  (make-readtable (current-readtable)
                  #\$ 'terminating-macro read-dollar))

(define read-dollar
  (case-lambda
    [(ch in)
     (check-$-after (arith:read in) in (object-name in))]
    [(ch in src line col pos)
     (check-$-after (arith:read-syntax src in) in src)]))

(define (check-$-after val in src)
  (regexp-match #px"^\\s*" in) ; skip whitespace
  (let ([ch (peek-char in)])
    (unless (equal? ch #\$) (bad-ending ch src in))
    (read-char in))
  val)

(define (bad-ending ch src in)
  (let-values ([(line col pos) (port-next-location in)])
```

```
((if (eof-object? ch)
     raise-read-error
     raise-read-eof-error)
 "expected a closing '$'"
 src line col pos
 (if (eof-object? ch) 0 1)))))
```

With this reader extension, a single `#reader` can be used at the beginning of an expression to enable multiple uses of `$` that switch to infix arithmetic:

```
> #reader"dollar.rkt" (let ([a $1*2+3$] [b $5/6$]) $a+b$)
35/6
```

17.3   Defining new `#lang` Languages

When loading a module as a source program that starts

```
#lang language
```

the *language* determines the way that the rest of the module is parsed at the reader level. The reader-level parse must produce a `module` form as a syntax object. As always, the second sub-form after `module` specifies the module language that controls the meaning of the module's body forms. Thus, a *language* specified after `#lang` controls both the reader-level and expander-level parsing of a module.

17.3.1   Designating a `#lang` Language

The syntax of a *language* intentionally overlaps with the syntax of a module path as used in `require` or as a module language, so that names like `racket`, `racket/base`, `slideshow`, or `scribble/manual` can be used both as `#lang` languages and as module paths.

At the same time, the syntax of *language* is far more restricted than a module path, because only `a-z`, `A-Z`, `0-9`, `/` (not at the start or end), `_`, `-`, and `+` are allowed in a *language* name. These restrictions keep the syntax of `#lang` as simple as possible. Keeping the syntax of `#lang` simple, in turn, is important because the syntax is inherently inflexible and non-extensible; the `#lang` protocol allows a *language* to refine and define syntax in a practically unconstrained way, but the `#lang` protocol itself must remain fixed so that various different tools can "boot" into the extended world.

Fortunately, the `#lang` protocol provides a natural way to refer to languages in ways other than the rigid *language* syntax: by defining a *language* that implements its own nested protocol. We have already seen one example (in §17.1.2 "Using `#lang s-exp`"): the `s-exp` *language* allows a programmer to specify a module language using the general module

path syntax. Meanwhile, `s-exp` takes care of the reader-level responsibilities of a #lang language.

Unlike `racket`, `s-exp` cannot be used as a module path with `require`. Although the syntax of *language* for #lang overlaps with the syntax of module paths, a *language* is not used directly as a module path. Instead, a *language* is suffixed with `/lang/reader` to obtain a module path, and the resulting module supplies `read` and `read-syntax` functions using a protocol that is similar to the one for `#reader`.

A consequence of the way that a #lang *language* is turned into a module path is that the language must be installed in a collection, similar to the way that `"racket"` or `"slideshow"` are collections that are distributed with Racket. Again, however, there's an escape from this restriction: the `reader` language lets you specify a reader-level implementation of a language using a general module path.

17.3.2 Using #lang `reader`

The `reader` language for #lang is similar to `s-exp`, in that it acts as a kind of meta-language. Whereas `s-exp` lets a programmer specify a module language at the expander layer of parsing, `reader` lets a programmer specify a language at the reader level.

A #lang `reader` must be followed by a module path, and the specified module must provide two functions: `read` and `read-syntax`. The protocol is the same as for a `#reader` implementation, but for #lang, the `read` and `read-syntax` functions must produce a `module` form that is based on the rest of the input file for the module.

The following `"literal.rkt"` module implements a language that treats its entire body as literal text and exports the text as a `data` string:

`"literal.rkt"`

```
#lang racket
(require syntax/strip-context)

(provide (rename-out [literal-read read]
                     [literal-read-syntax read-syntax]))

(define (literal-read in)
  (syntax->datum
   (literal-read-syntax #f in)))

(define (literal-read-syntax src in)
  (with-syntax ([str (port->string in)])
    (strip-context
     #'(module anything racket
         (provide data)
```

273

```
            (define data 'str)))))
```

The `"literal.rkt"` language uses `strip-context` on the generated `module` expression, because a `read-syntax` function should return a syntax obejct with no lexical context. Also, the `"literal.rkt"` language creates a module named `anything`, which is an arbitrary choice; the language is intended to be used in a file, and the longhand module name is ignored when it appears in a `required` file.

The `"literal.rkt"` language can be used in a module `"tuvalu.rkt"`:

> `"tuvalu.rkt"`

```
#lang reader "literal.rkt"
Technology!
System!
Perfect!
```

Importing `"tuvalu.rkt"` binds `data` to a string version of the module content:

```
> (require "tuvalu.rkt")
> data
"\nTechnology!\nSystem!\nPerfect!\n"
```

17.3.3 Using `#lang s-exp syntax/module-reader`

Parsing a module body is usually not as trivial as in `"literal.rkt"`. A more typical module parser must iterate to parse multiple forms for a module body. A language is also more likely to extend Racket syntax—perhaps through a readtable—instead of replacing Racket syntax completely.

The `syntax/module-reader` module language abstracts over common parts of a language implementation to simplify the creation of new languages. In its most basic form, a language implemented with `syntax/module-reader` simply specifies the module language to be used for the language, in which case the reader layer of the language is the same as Racket. For example, with

> `"raquet-mlang.rkt"`

```
#lang racket
(provide (except-out (all-from-out racket) lambda)
         (rename-out [lambda function]))
```

and

> `"raquet.rkt"`

```
#lang s-exp syntax/module-reader
"raquet-mlang.rkt"
```

274

then

```
#lang reader "raquet.rkt"
(define identity (function (x) x))
(provide identity)
```

implements and exports the `identity` function, since `"raquet-mlang.rkt"` exports `lambda` as `function`.

The `syntax/module-reader` language accepts many optional specifications to adjust other features of the language. For example, an alternate `read` and `read-syntax` for parsing the language can be spcified with `#:read` and `#:read-syntax`, respectively. The following `"dollar-racket.rkt"` language uses `"dollar.rkt"` (see §17.2.2 "Readtables") to build a language that is like `racket` but with a `$` escape to simple infix arithmetic:

`"dollar-racket.rkt"`

```
#lang s-exp syntax/module-reader
racket
#:read $-read
#:read-syntax $-read-syntax

(require (prefix-in $- "dollar.rkt"))
```

The `require` form appears at the end of the module, because all of the keyword-tagged optional specifications for `syntax/module-reader` must appear before any helper imports or definitions.

The following module uses `"dollar-racket.rkt"` to implement a `cost` function using a `$` escape:

`"store.rkt"`

```
#lang reader "dollar-racket.rkt"

(provide cost)

; Cost of 'n' $1 rackets with 7% sales
; tax and shipping-and-handling fee 'h':
(define (cost n h)
  $n*107/100+h$)
```

17.3.4   Installing a Language

So far, we have used the `reader` meta-language to access languages like `"literal.rkt"` and `"dollar-racket.rkt"`. If you want to use something like `#lang literal` directly, then you must move `"literal.rkt"` into a Racket collection named `"literal"`.

To install a collection, you can create a directory either in the main Racket installation or in a user-specific directory. Use `find-collects-dir` or `find-user-collects-dir` from `setup/dirs` to find the directory:

```
> (require setup/dirs)
> (find-user-collects-dir)
#<path:/home/racketeer/.racket/5.1/collects>
```

Move `"literal.rkt"` to `"literal/lang/reader.rkt"` within the directory reported by `find-collects-dir` or `find-user-collects-dir`. That is, the file `"literal.rkt"` must be renamed to `"reader.rkt"` and placed in a `"lang"` sub-directory of the `"literal"` collection.

```
.... (the main installation or the user's space)
 |- "collects"
      |- "literal"
           |- "lang"
                |- "reader.rkt"
```

After moving the file, you can use `literal` directly after `#lang`:

```
#lang literal
Technology!
System!
Perfect!
```

You can also package a collection for others to install by using the `raco pack` command-line tool:

See *raco: Racket Command-Line Tools* for more information on using `raco`.

```
raco pack --collection literal.plt literal
```

Then, others can install the `"literal"` collection using `raco setup`:

```
raco setup literal.plt
```

See **PLaneT**: *Automatic Package Distribution* for more information about PLaneT packages.

A better approach may be to distribute your language as a PLaneT package. A drawback of using a PLaneT package is that users must type `#lang planet` followed by a PLaneT path to access the language. The great advantages are that the PLaneT package can be installed automatically, it can be versioned, and it co-exists more easily with other packages.

17.3.5 Source-Handling Configuration

The Racket distribution includes a Scribble language for writing prose documents, where Scribble extends the normal Racket to better support text. Here is an example Scribble document:

```
#lang scribble/base

@(define (get-name) "Self-Describing Document")

@title[(get-name)]

The title of this document is ''@(get-name).''
```

If you put that program in DrRacket's definitions area and click Run, then nothing much appears to happen. The scribble/base language just binds and exports doc as a description of a document, similar to the way that "literal.rkt" exports a string as data.

Simply opening a module with the language scribble/base in DrRacket, however, causes a Scribble HTML button to appear. Furthermore, DrRacket knows how to colorize Scribble syntax by coloring green those parts of the document that correspond to literal text. The language name scribble/base is not hard-wired into DrRacket. Instead, the implementation of the scribble/base language provides button and syntax-coloring information in response to a query from DrRacket.

For security reasons, only languages that have been specifically installed by a user can respond to language-information queries. If you have installed the literal language as described in §17.3.4 "Installing a Language", then you can adjust "literal/lang/reader.rkt" so that DrRacket treats the content of a module in the literal language as plain text instead of (erroneously) as Racket syntax:

"literal/lang/reader.rkt"

```
#lang racket
(require syntax/strip-context)

(provide (rename-out [literal-read read]
                     [literal-read-syntax read-syntax])
         get-info)

(define (literal-read in)
  (syntax->datum
   (literal-read-syntax #f in)))

(define (literal-read-syntax src in)
  (with-syntax ([str (port->string in)])
    (strip-context
     #'(module anything racket
         (provide data)
         (define data 'str)))))

(define (get-info in mod line col pos)
  (lambda (key default)
```

277

```
(case key
  [(color-lexer)
   (dynamic-require 'syntax-color/default-lexer
                    'default-lexer)]
  [else default])))
```

This revised `literal` implementation provides a `get-info` function. The `get-info` function will be applied to the source input stream and location information, in case query results should depend on the content of the module after the language name (which is not the case for `literal`). The result of `get-info` is a function of two arguments. The first argument is always a symbol, indicating the kind of information that a tool requests from the language; the second argument is the default result to be returned if the language does not recognize the query or has no information for it.

After DrRacket obtains the result of `get-info` for a language, it calls the function with a `'color-lexer` query; the result should be a function that implements syntax-coloring parsing on an input stream. For `literal`, the `syntax-color/default-lexer` module provides a `default-lexer` syntax-coloring parser that is suitable for plain text, so `literal` loads and returns that parser in response to a `'color-lexer` query.

The set of symbols that a programming tool uses for queries is entirely between the tool and the languages that choose to cooperate with it. For example, in addition to `'color-lexer`, DrRacket uses a `'drscheme:toolbar-buttons` query to determine which buttons should be available in the toolbar to operate on modules using the language.

The `syntax/module-reader` language lets you specify `get-info` handling through a `#:info` optional specification. The protocol for an `#:info` function is slightly different from the raw `get-info` protocol; the revised protocol allows `syntax/module-reader` the possibility of handling future language-information queries automatically.

17.3.6 Module-Handling Configuration

Suppose that the file `"death-list-5.rkt"` contains

`"death-list-5.rkt"`

```
#lang racket
(list "O-Ren Ishii"
      "Vernita Green"
      "Budd"
      "Elle Driver"
      "Bill")
```

If you `require` `"death-list-5.rkt"` directly, then it prints the list in the usual Racket result format:

```
> (require "death-list-5.rkt")
'("O-Ren Ishii" "Vernita Green" "Budd" "Elle Driver" "Bill")
```

However, if `"death-list-5.rkt"` is required by a `"kiddo.rkt"` that is implemented with `scheme` instead of `racket`:

```
                                                            "kiddo.rkt"
 #lang scheme
 (require "death-list-5.rkt")
```

then, if you run `"kiddo.rkt"` file in DrRacket or if you run it directly with `racket`, `"kiddo.rkt"` causes `"death-list-5.rkt"` to print its list in traditional Scheme format, without the leading quote:

```
 ("O-Ren Ishii" "Vernita Green" "Budd" "Elle Driver" "Bill")
```

The `"kiddo.rkt"` example illustrates how the format for printing a result value can depend on the main module of a program instead of the language that is used to implement it.

Unlike the syntax-coloring property of a language (as described in §17.3.5 "Source-Handling Configuration"), the result-value format is a property of a *module* (via its language) as opposed to a property of the module's *source text*. That is, the run-time configuration for a module should be available even if the module is compiled to bytecode form and the source is unavailable. Due to this difference, language properties such as run-time configuration are not reported via a `get-info` function that exported from the language's parser module, but instead through a separate module whose name is attached to the syntax object for a parsed `module` form.

Going back to the `literal` language (see §17.3.5 "Source-Handling Configuration"), we can adjust the language so that directly running a `literal` module causes it to print out its string, while using a `literal` module in a larger program simply provides `data` without printing. To make this work, we will need three extra module files:

```
 ....  (the main installation or the user's space)
 |- "collects"
     |- "literal"
         |- "lang"
         |     |- "reader.rkt"
         |- "language-info.rkt"    (new)
         |- "runtime-config.rkt"   (new)
         |- "show.rkt"             (new)
```

- The `"literal/language-info.rkt"` module provides reflective information about the language of modules written in the `literal` language. The name of this module is not special; it will be connected to the `literal` language through a change to `"literal/lang/reader.rkt"`.

- The "literal/runtime-config.rkt" module will be identified by "literal/language-info.rkt" as the run-time configuration code for a main module that uses the literal language.

- The "literal/show.rkt" module will provide a show function to be applied to the string content of a literal module. The run-time configuration action in "literal/runtime-config.rkt" will instruct show to print the strings that it is given, but only when a module using the literal language is run directly.

Multiple modules are needed to implement the printing change, because the different modules must run at different times. For example, the code needed to parse a literal module is not needed after the module has been compiled, while the run-time configuration code is needed only when the module is run as the main module of a program. Similarly, when creating a stand-alone executable with raco exe, the main module (in compiled form) must be queried for its run-time configuration, but the module and its configuration action should not run until the executable is started. By using different modules for these different tasks, we avoid loading code at times when it is not needed.

The three new files are connected to the literal language by changes to "literal/lang/reader.rkt":

- The module form generated by the read-syntax function must import the literal/show module and call its show function.

- The module form must be annotated with a 'language-info syntax property, whose value points to a get-language-info function exported by a literal/language-info module. The get-language-info function will be responsible for reporting the literal/runtime-config as the run-time configuration action of the language.

  The 'language-info syntax property value is a vector that contains a module (in this case literal/language-info), a symbol for one of the module's exports (get-language-info in this case), and an data value (which is not needed in this case). The data component allows information to be propagated from the source to the module's language information.

These changes are implemented in the following revised "literal/lang/reader.rkt":

"literal/lang/reader.rkt"

```
#lang racket
(require syntax/strip-context)

(provide (rename-out [literal-read read]
                     [literal-read-syntax read-syntax])
         get-info)

(define (literal-read in)
```

280

```
      (syntax->datum
       (literal-read-syntax #f in)))

  (define (literal-read-syntax src in)
    (with-syntax ([str (port->string in)])
      (syntax-property
       (strip-context
        #'(module anything racket
            (require literal/show)
            (provide data)
            (define data 'str)
            (show data)))
       'module-language
       '#(literal/language-info get-language-info #f))))

  (define (get-info in mod line col pos)
    (lambda (key default)
      (case key
        [(color-lexer)
         (dynamic-require 'syntax-color/default-lexer
                          'default-lexer)]
        [else default])))
```

When a `module` form with a `'module-language` property is compiled, the property value is preserved with the compiled module, and it is accessible via reflective functions like `module->language-info`. When `racket` or DrRacket runs a module, it uses `module->language-info` to obtain a vector that contains a module name, export name, and data value. The result of the function applied to the data should be another function that answers queries, much like the `get-info` function in a language reader.

For `literal`, `"literal/language-info.rkt"` is implemented as:

```
                                          "literal/language-info.rkt"

  #lang racket

  (provide get-language-info)

  (define (get-language-info data)
    (lambda (key default)
      (case key
        [(configure-runtime)
         '(#(literal/runtime-config configure #f))]
        [else default])))
```

The function returned by `get-language-info` answers a `'configure-runtime` query with a list of yet more vectors, where each vector contains a module name, an exported name,

and a data value. For the `literal` language, the run-time configuration action implemented in `"literal/runtime-config.rkt"` is to enable printing of strings that are sent to `show`:

`"literal/runtime-config.rkt"`

```racket
#lang racket
(require "show.rkt")

(provide configure)

(define (configure data)
  (show-enabled #t))
```

Finally, the `"literal/runtime-config.rkt"` module must provide the `show-enabled` parameter and `show` function:

`"literal/runtime-config.rkt"`

```racket
#lang racket

(provide show show-enabled)

(define show-enabled (make-parameter #f))

(define (show v)
  (when (show-enabled)
    (display v)))
```

With all of the pieces for `literal` in place, try running the following variant of `"tuvalu.rkt"` directly and through a `require` from another module:

`"tuvalu.rkt"`

```
#lang literal
Technology!
System!
Perfect!
```

When using `syntax/module-reader` to implement a language, specify a module's language information through the `#:language-info` optional specification. The value provided through `#:language-info` is attached to a `module` form directly as a syntax property.

282

18 Performance

Alan Perlis famously quipped "Lisp programmers know the value of everything and the cost of nothing." A Racket programmer knows, for example, that a `lambda` anywhere in a program produces a value that is closed over its lexical environment—but how much does allocating that value cost? While most programmers have a reasonable grasp of the cost of various operations and data structures at the machine level, the gap between the Racket language model and the underlying computing machinery can be quite large.

In this chapter, we narrow the gap by explaining details of the Racket compiler and run-time system and how they affect the run-time and memory performance of Racket code.

18.1 The Bytecode and Just-in-Time (JIT) Compilers

Every definition or expression to be evaluated by Racket is compiled to an internal bytecode format. In interactive mode, this compilation occurs automatically and on-the-fly. Tools like `raco make` and `raco setup` marshal compiled bytecode to a file, so that you do not have to compile from source every time that you run a program. (Most of the time required to compile a file is actually in macro expansion; generating bytecode from fully expanded code is relatively fast.) See §20 "Compilation and Configuration" for more information on generating bytecode files.

The bytecode compiler applies all standard optimizations, such as constant propagation, constant folding, inlining, and dead-code elimination. For example, in an environment where `+` has its usual binding, the expression `(let ([x 1] [y (lambda () 4)]) (+ 1 (y)))` is compiled the same as the constant `5`.

On some platforms, bytecode is further compiled to native code via a *just-in-time* or *JIT* compiler. The JIT compiler substantially speeds programs that execute tight loops, arithmetic on small integers, and arithmetic on inexact real numbers. Currently, JIT compilation is supported for x86, x86_64 (a.k.a. AMD64), and 32-bit PowerPC processors. The JIT compiler can be disabled via the `eval-jit-enabled` parameter or the `--no-jit`/`-j` command-line flag for `racket`.

The JIT compiler works incrementally as functions are applied, but the JIT compiler makes only limited use of run-time information when compiling procedures, since the code for a given module body or `lambda` abstraction is compiled only once. The JIT's granularity of compilation is a single procedure body, not counting the bodies of any lexically nested procedures. The overhead for JIT compilation is normally so small that it is difficult to detect.

18.2 Modules and Performance

The module system aids optimization by helping to ensure that identifiers have the usual bindings. That is, the `+` provided by `racket/base` can be recognized by the compiler and inlined, which is especially important for JIT-compiled code. In contrast, in a traditional interactive Scheme system, the top-level `+` binding might be redefined, so the compiler cannot assume a fixed `+` binding (unless special flags or declarations are used to compensate for the lack of a module system).

Even in the top-level environment, importing with `require` enables some inlining optimizations. Although a `+` definition at the top level might shadow an imported `+`, the shadowing definition applies only to expressions evaluated later.

Within a module, inlining and constant-propagation optimizations take additional advantage of the fact that definitions within a module cannot be mutated when no `set!` is visable at compile time. Such optimizations are unavailable in the top-level environment. Although this optimization within modules is important for performance, it hinders some forms of interactive development and exploration. The `compile-enforce-module-constants` parameter disables the JIT compiler's assumptions about module definitions when interactive exploration is more important. See §6.6 "Assignment and Redefinition" for more information.

Currently, the compiler does not attempt to inline or propagate constants across module boundaries, except for exports of the built-in modules (such as the one that originally provides `+`).

The later section §18.5 "`letrec` Performance" provides some additional caveats concerning inlining of module bindings.

18.3 Function-Call Optimizations

When the compiler detects a function call to an immediately visible function, it generates more efficient code than for a generic call, especially for tail calls. For example, given the program

```
(letrec ([odd (lambda (x)
                (if (zero? x)
                    #f
                    (even (sub1 x))))]
         [even (lambda (x)
                 (if (zero? x)
                     #t
                     (odd (sub1 x))))])
  (odd 40000000))
```

the compiler can detect the odd–even loop and produce code that runs much faster via loop unrolling and related optimizations.

Within a module form, defined variables are lexically scoped like letrec bindings, and definitions within a module therefore permit call optimizations, so

```
(define (odd x) ....)
(define (even x) ....)
```

within a module would perform the same as the letrec version.

Primitive operations like pair?, car, and cdr are inlined at the machine-code level by the JIT compiler. See also the later section §18.6 "Fixnum and Flonum Optimizations" for information about inlined arithmetic operations.

18.4 Mutation and Performance

Using set! to mutate a variable can lead to bad performance. For example, the microbenchmark

```
#lang racket/base

(define (subtract-one x)
  (set! x (sub1 x))
  x)

(time
  (let loop ([n 4000000])
    (if (zero? n)
        'done
        (loop (subtract-one n)))))
```

runs much more slowly than the equivalent

```
#lang racket/base

(define (subtract-one x)
  (sub1 x))

(time
  (let loop ([n 4000000])
    (if (zero? n)
        'done
        (loop (subtract-one n)))))
```

285

In the first variant, a new location is allocated for `x` on every iteration, leading to poor performance. A more clever compiler could unravel the use of `set!` in the first example, but since mutation is discouraged (see §4.9.1 "Guidelines for Using Assignment"), the compiler's effort is spent elsewhere.

More significantly, mutation can obscure bindings where inlining and constant-propagation might otherwise apply. For example, in

```
(let ([minus1 #f])
  (set! minus1 sub1)
  (let loop ([n 4000000])
    (if (zero? n)
        'done
        (loop (minus1 n)))))
```

the `set!` obscures the fact that `minus1` is just another name for the built-in `sub1`.

18.5 `letrec` Performance

When `letrec` is used to bind only procedures and literals, then the compiler can treat the bindings in an optimal manner, compiling uses of the bindings efficiently. When other kinds of bindings are mixed with procedures, the compiler may be less able to determine the control flow.

For example,

```
(letrec ([loop (lambda (x)
                 (if (zero? x)
                     'done
                     (loop (next x))))]
         [junk (display loop)]
         [next (lambda (x) (sub1 x))])
  (loop 40000000))
```

likely compiles to less efficient code than

```
(letrec ([loop (lambda (x)
                 (if (zero? x)
                     'done
                     (loop (next x))))]
         [next (lambda (x) (sub1 x))])
  (loop 40000000))
```

In the first case, the compiler likely does not know that `display` does not call `loop`. If it did, then `loop` might refer to `next` before the binding is available.

This caveat about `letrec` also applies to definitions of functions and constants as internal definitions or in modules. A definition sequence in a module body is analogous to a sequence of `letrec` bindings, and non-constant expressions in a module body can interfere with the optimization of references to later bindings.

18.6 Fixnum and Flonum Optimizations

A *fixnum* is a small exact integer. In this case, "small" depends on the platform. For a 32-bit machine, numbers that can be expressed in 30 bits plus a sign bit are represented as fixnums. On a 64-bit machine, 62 bits plus a sign bit are available.

A *flonum* is used to represent any inexact real number. They correspond to 64-bit IEEE floating-point numbers on all platforms.

Inlined fixnum and flonum arithmetic operations are among the most important advantages of the JIT compiler. For example, when + is applied to two arguments, the generated machine code tests whether the two arguments are fixnums, and if so, it uses the machine's instruction to add the numbers (and check for overflow). If the two numbers are not fixnums, then it checks whether whether both are flonums; in that case, the machine's floating-point operations are used directly. For functions that take any number of arguments, such as +, inlining works for two or more arguments (except for -, whose one-argument case is also inlined) when the arguments are either all fixnums or all flonums.

Flonums are typically *boxed*, which means that memory is allocated to hold every result of a flonum computation. Fortunately, the generational garbage collector (described later in §18.8 "Memory Management") makes allocation for short-lived results reasonably cheap. Fixnums, in contrast are never boxed, so they are typically cheap to use.

See §18.9 "Parallelism with Futures" for an example use of flonum-specific operations.

The `racket/flonum` library provides flonum-specific operations, and combinations of flonum operations allow the JIT compiler to generate code that avoids boxing and unboxing intermediate results. Besides results within immediate combinations, flonum-specific results that are bound with `let` and consumed by a later flonum-specific operation are unboxed within temporary storage. Finally, the compiler can detect some flonum-valued loop accumulators and avoid boxing of the accumulator. The bytecode decompiler (see §7 "`raco de-compile`: Decompiling Bytecode") annotates combinations where the JIT can avoid boxes with `#%flonum`, `#%as-flonum`, and `#%from-flonum`.

Unboxing of local bindings and accumualtors is not supported by the JIT for PowerPC.

The `racket/unsafe/ops` library provides unchecked fixnum- and flonum-specific operations. Unchecked flonum-specific operations allow unboxing, and sometimes they allow the compiler to reorder expressions to improve performance. See also §18.7 "Unchecked, Unsafe Operations", especially the warnings about unsafety.

18.7   Unchecked, Unsafe Operations

The `racket/unsafe/ops` library provides functions that are like other functions in `racket/base`, but they assume (instead of checking) that provided arguments are of the right type. For example, `unsafe-vector-ref` accesses an element from a vector without checking that its first argument is actually a vector and without checking that the given index is in bounds. For tight loops that use these functions, avoiding checks can sometimes speed the computation, though the benefits vary for different unchecked functions and different contexts.

Beware that, as "unsafe" in the library and function names suggest, misusing the exports of `racket/unsafe/ops` can lead to crashes or memory corruption.

18.8   Memory Management

The Racket implementation is available in two variants: *3m* and *CGC*. The 3m variant uses a modern, *generational garbage collector* that makes allocation relatively cheap for short-lived objects. The CGC variant uses a *conservative garbage collector* which facilitates interaction with C code at the expense of both precision and speed for Racket memory management. The 3m variant is the standard one.

Although memory allocation is reasonably cheap, avoiding allocation altogether is normally faster. One particular place where allocation can be avoided sometimes is in *closures*, which are the run-time representation of functions that contain free variables. For example,

```
(let loop ([n 40000000] [prev-thunk (lambda () #f)])
  (if (zero? n)
      (prev-thunk)
      (loop (sub1 n)
            (lambda () n))))
```

allocates a closure on every iteration, since `(lambda () n)` effectively saves `n`.

The compiler can eliminate many closures automatically. For example, in

```
(let loop ([n 40000000] [prev-val #f])
  (let ([prev-thunk (lambda () n)])
    (if (zero? n)
        prev-val
        (loop (sub1 n) (prev-thunk)))))
```

no closure is ever allocated for `prev-thunk`, because its only application is visible, and so it is inlined. Similarly, in

```
(let n-loop ([n 400000])
```

```
(if (zero? n)
    'done
    (let m-loop ([m 100])
      (if (zero? m)
          (n-loop (sub1 n))
          (m-loop (sub1 m))))))))
```

then the expansion of the `let` form to implement `m-loop` involves a closure over `n`, but the compiler automatically converts the closure to pass itself `n` as an argument instead.

18.9 Parallelism with Futures

The `racket/future` library provides support for performance improvement through parallelism with the `future` and `touch` functions. The level of parallelism available from those constructs, however, is limited by several factors, and the current implementation is best suited to numerical tasks.

As a starting example, the `any-double?` function below takes a list of numbers and determines whether any number in the list has a double that is also in the list:

```
(define (any-double? l)
  (for/or ([i (in-list l)])
    (for/or ([i2 (in-list l)])
      (= i2 (* 2 i)))))
```

This function runs in quadratic time, so it can take a long time (on the order of a second) on large lists like `l1` and `l2`:

```
(define l1 (for/list ([i (in-range 5000)])
             (+ (* 2 i) 1)))
(define l2 (for/list ([i (in-range 5000)])
             (- (* 2 i) 1)))
(or (any-double? l1)
    (any-double? l2))
```

The best way to speed up `any-double?` is to use a different algorithm. However, on a machine that offers at least two processing units, the example above can run in about half the time using `future` and `touch`:

```
(let ([f (future (lambda () (any-double? l2)))])
  (or (any-double? l1)
      (touch f)))
```

The future `f` runs `(any-double? l2)` in parallel to `(any-double? l1)`, and the result for `(any-double? l2)` becomes available about the same time that it is demanded by `(touch`

Other functions, such as `thread`, support the creation of reliably concurrent tasks. However, thread never run truly in parallel, even if the hardware and operating system support parallelism.

289

```
f).
```

Futures run in parallel as long as they can do so safely, but the notion of "safe" for parallelism is inherently tied to the system implementation. The distinction between "safe" and "unsafe" operations may be far from apparent at the level of a Racket program.

Consider the following core of a Mandelbrot-set computation:

```
(define (mandelbrot iterations x y n)
  (let ((ci (- (/ (* 2.0 y) n) 1.0))
        (cr (- (/ (* 2.0 x) n) 1.5)))
    (let loop ((i 0) (zr 0.0) (zi 0.0))
      (if (> i iterations)
          i
          (let ((zrq (* zr zr))
                (ziq (* zi zi)))
            (cond
              ((> (+ zrq ziq) 4.0) i)
              (else (loop (add1 i)
                          (+ (- zrq ziq) cr)
                          (+ (* 2.0 zr zi) ci)))))))))
```

The expressions `(mandelbrot 10000000 62 500 1000)` and `(mandelbrot 10000000 62 501 1000)` each take a while to produce an answer. Computing them both, of course, takes twice as long:

```
(list (mandelbrot 10000000 62 500 1000)
      (mandelbrot 10000000 62 501 1000))
```

Unfortunately, attempting to run the two computations in parallel with `future` does not improve performance:

```
(let ([f (future (lambda () (mandelbrot 10000000 62 501 1000)))])
  (list (mandelbrot 10000000 62 500 1000)
        (touch f)))
```

One problem is that the `*` and `/` operations in the first two lines of `mandelbrot` involve a mixture of exact and inexact real numbers. Such mixtures typically trigger a slow path in execution, and the general slow path is not safe for parallelism. Consequently, the future created in this example is almost immediately suspended, and it cannot resume until `touch` is called.

Changing the first two lines of `mandelbrot` addresses that first the problem:

```
(define (mandelbrot iterations x y n)
  (let ((ci (- (/ (* 2.0 (->fl y)) (->fl n)) 1.0))
        (cr (- (/ (* 2.0 (->fl x)) (->fl n)) 1.5)))
```

```
        ....))
```

With that change, `mandelbrot` computations can run in parallel. Nevertheless, performance still does not improve. The problem is that most every arithmetic operation in this example produces an inexact number whose storage must be allocated. Especially frequent allocation triggers communication between parallel tasks that defeats any performance improvement.

By using flonum-specific operations (see §18.6 "Fixnum and Flonum Optimizations"), we can re-write `mandelbot` to use much less allocation:

```
(define (mandelbrot iterations x y n)
  (let ((ci (fl- (fl/ (* 2.0 (->fl y)) (->fl n)) 1.0))
        (cr (fl- (fl/ (* 2.0 (->fl x)) (->fl n)) 1.5)))
    (let loop ((i 0) (zr 0.0) (zi 0.0))
      (if (> i iterations)
          i
          (let ((zrq (fl* zr zr))
                (ziq (fl* zi zi)))
            (cond
             ((fl> (fl+ zrq ziq) 4.0) i)
             (else (loop (add1 i)
                         (fl+ (fl- zrq ziq) cr)
                         (fl+ (fl* 2.0 (fl* zr zi)) ci)))))))))
```

This conversion can speed `mandelbrot` by a factor of 8, even in sequential mode, but avoiding allocation also allows `mandelbrot` to run usefully faster in parallel.

As a general guideline, any operation that is inlined by the JIT compiler runs safely in parallel, while other operations that are not inlined (including all operations if the JIT compiler is disabled) are considered unsafe. The `mzc` decompiler tool annotates operations that can be inlined by the compiler (see §7 "`raco decompile`: Decompiling Bytecode"), so the decompiler can be used to help predict parallel performance.

To more directly report what is happening in a program that uses `future` and `touch`, operations are logged when they suspend a computation or synchronize with the main computation. For example, running the original `mandelbrot` in a future produces the following output in the `'debug` log level:

```
  future: 0 waiting for runtime at 1267392979341.989: *
```

The message indicates which internal future-running task became blocked on an unsafe operation, the time it blocked (in terms of `current-inexact-miliseconds`), and the operation that caused the computation it to block.

The first revision to `mandelbrot` avoids suspending at `*`, but produces many log entries of the form

```
future: 0 waiting for runtime at 1267392980465.066: [ac-
quire_gc_page]
```

19   Running and Creating Executables

While developing programs, many Racket programmers use the DrRacket programming environment. To run a program without the development environment, use `racket` (for console-based programs) or `gracket` (for GUI programs). This chapter mainly explains how to run `racket` and `gracket`.

19.1   Running `racket` and `gracket`

The `gracket` executable is the same as `racket`, but with small adjustments to behave as a GUI application rather than a console application. For example, `gracket` by default runs in interactive mode with a GUI window instead of a console prompt. GUI applications can be run with plain `racket`, however.

Depending on command-line arguments, `racket` or `gracket` runs in interactive mode, module mode, or load mode.

19.1.1   Interactive Mode

When `racket` is run with no command-line arguments (other than confguration options, like `-j`), then it starts a REPL with a `>` prompt:

```
Welcome to Racket
>
```

To initialize the REPL's environment, `racket` first requires the `racket/init` module, which provides all of `racket`, and also installs `pretty-print` for display results. Finally, `racket` loads the file reported by (`find-system-path` `'init-file`), if it exists, before starting the REPL.

For information on GNU Readline support, see `readline`.

If any command-line arguments are provided (other than configuration options), add `-i` or `--repl` to re-enable the REPL. For example,

```
racket -e '(display "hi\n")' -i
```

displays "hi" on start-up, but still presents a REPL.

If module-requiring flags appear before `-i`/`--repl`, they cancel the automatic requiring of `racket/init`. This behavior can be used to initialize the REPL's environment with a different language. For example,

```
racket -l racket/base -i
```

starts a REPL using a much smaller initial language (that loads much faster). Beware that most modules do not provide the basic syntax of Racket, including function-call syntax and `require`. For example,

```
racket -l racket/date -i
```

produces a REPL that fails for every expression, because `racket/date` provides only a few functions, and not the `#%top-interaction` and `#%app` bindings that are needed to evaluate top-level function calls in the REPL.

If a module-requiring flag appears after `-i/--repl` instead of before it, then the module is required after `racket/init` to augment the initial environment. For example,

```
racket -i -l racket/date
```

starts a useful REPL with `racket/date` available in addition to the exports of `racket`.

19.1.2 Module Mode

If a file argument is supplied to `racket` before any command-line switch (other than configuration options), then the file is required as a module, and (unless `-i/--repl` is specified), no REPL is started. For example,

```
racket hello.rkt
```

requires the `"hello.rkt"` module and then exits. Any argument after the file name, flag or otherwise, is preserved as a command-line argument for use by the required module via `current-command-line-arguments`.

If command-line flags are used, then the -u or `--require-script` flag can be used to explicitly require a file as a module. The `-t` or `--require` flag is similar, except that additional command-line flags are processed by `racket`, instead of preserved for the required module. For example,

```
racket -t hello.rkt -t goodbye.rkt
```

requires the `"hello.rkt"` module, then requires the `"goodbye.rkt"` module, and then exits.

The `-l` or `--lib` flag is similar to `-t/--require`, but it requires a module using a `lib` module path instead of a file path. For example,

```
racket -l raco
```

is the same as running the `raco` executable with no arguments, since the `raco` module is the executable's main module.

Note that if you wanted to pass command-line flags to `raco` above, you would need to protect the flags with a `--`, so that `racket` doesn't try to parse them itself:

```
racket -l raco -- --help
```

19.1.3 Load Mode

The `-f` or `--load` flag supports `load`ing top-level expressions in a file directly, as opposed to expressions within a module file. This evaluation is like starting a REPL and typing the expressions directly, except that the results are not printed. For example,

```
racket -f hi.rkts
```

`load`s "hi.rkts" and exits. Note that load mode is generally a bad idea, for the reasons explained in §1.4 "A Note to Readers with Lisp/Scheme Experience"; using module mode is typically better.

The `-e` or `--eval` flag accepts an expression to evaluate directly. Unlike file loading, the result of the expression is printed, as in a REPL. For example,

```
racket -e '(current-seconds)'
```

prints the number of seconds since January 1, 1970.

For file loading and expression evaluation, the top-level environment is created in the same way for interactive mode: `racket/init` is required unless another module is specified first. For example,

```
racket -l racket/base -e '(current-seconds)'
```

likely runs faster, because it initializes the environment for evaluation using the smaller `racket/base` language, instead of `racket/init`.

19.2 Scripts

Racket files can be turned into executable scripts under Unix and Mac OS X. Under Windows, a compatibility layer like Cygwin support the same kind of scripts, or scripts can be implemented as batch files.

19.2.1 Unix Scripts

In a Unix environment (including Linux and Mac OS X), a Racket file can be turned into an executable script using the shell's `#!` convention. The first two characters of the file must be

`#!`; the next character must be either a space or `/`, and the remainder of the first line must be a command to execute the script. For some platforms, the total length of the first line is restricted to 32 characters, and sometimes the space is required.

The simplest script format uses an absolute path to a `racket` executable followed by a module declaration. For example, if `racket` is installed in `"/usr/local/bin"`, then a file containing the following text acts as a "hello world" script:

```
#! /usr/local/bin/racket
#lang racket/base
"Hello, world!"
```

In particular, if the above is put into a file `"hello"` and the file is made executable (e.g., with `chmod a+x hello`), then typing `./hello` at the shell prompt produces the output `"Hello, world!"`.

The above script works because the operating system automatically puts the path to the script as the argument to the program started by the `#!` line, and because `racket` treats a single non-flag argument as a file containing a module to run.

Instead of specifying a complete path to the `racket` executable, a popular alternative is to require that `racket` is in the user's command path, and then "trampoline" using `/usr/bin/env`:

```
#! /usr/bin/env racket
#lang racket/base
"Hello, world!"
```

In either case, command-line arguments to a script are available via `current-command-line-arguments`:

```
#! /usr/bin/env racket
#lang racket/base
(printf "Given arguments: ~s\n"
        (current-command-line-arguments))
```

If the name of the script is needed, it is available via `(find-system-path 'run-file)`, instead of via `(current-command-line-arguments)`.

Usually, then best way to handle command-line arguments is to parse them using the `command-line` form provided by `racket`. The `command-line` form extracts command-line arguments from `(current-command-line-arguments)` by default:

```
#! /usr/bin/env racket
#lang racket

(define verbose? (make-parameter #f))
```

```
(define greeting
  (command-line
   #:once-each
   [("-v") "Verbose mode" (verbose? #t)]
   #:args
   (str) str))

(printf "~a~a\n"
        greeting
        (if (verbose?) " to you, too!" ""))
```

Try running the above script with the `--help` flag to see what command-line arguments are allowed by the script.

An even more general trampoline uses `/bin/sh` plus some lines that are comments in one language and expressions in the other. This trampoline is more complicated, but it provides more control over command-line arguments to `racket`:

```
#! /bin/sh
#|
exec racket -cu "$0" ${1+"$@"}
|#
#lang racket/base
(printf "This script started slowly, because the use of\n")
(printf "bytecode files has been disabled via -c.\n")
(printf "Given arguments: ~s\n"
        (current-command-line-arguments))
```

Note that `#!` starts a line comment in Racket, and `#|`...`|#` forms a block comment. Meanwhile, `#` also starts a shell-script comment, while `exec racket` aborts the shell script to start `racket`. That way, the script file turns out to be valid input to both `/bin/sh` and `racket`.

19.2.2 Windows Batch Files

A similar trick can be used to write Racket code in Windows `.bat` batch files:

```
; @echo off
; Racket.exe "%~f0" %*
; exit /b
#lang racket/base
"Hello, world!"
```

19.3 Creating Stand-Alone Executables

For information on creating and distributing executables, see §2 "`raco exe`: Creating Stand-Alone Executables" and §3 "`raco distribute`: Sharing Stand-Alone Executables" in *`raco`: Racket Command-Line Tools*.

20 Compilation and Configuration

So far in this guide, we have mainly discussed DrRacket and `racket` (and `gracket`). The main additional executable is `raco`, which is short for "**Ra**cket **co**mmand." The `raco` program provides a command-line interface to many additional tools for compiling Racket programs and maintaining a Racket installation.

- `raco make` compiles Racket source to bytecode.

  For example, if you have a program `"take-over-world.rkt"` and you'd like to compile it to bytecode, along with all of its dependencies, so that it loads more quickly, then run

  ```
  raco make take-over-the-world.rkt
  ```

  The bytecode file is written as `"take-over-the-world_rkt.zo"` in a `"compiled"` subdirectory; `".zo"` is the file suffix for a bytecode file.

- `raco setup` manages a Racket installation, including manually installed packages.

  For example, if you create your own library collection called `"take-over"`, and you'd like to build all bytecode and documentation for the collection, then run

  ```
  raco setup -l take-over
  ```

- `raco planet` manages packages that are normally downloaded automatically, on demand.

  For example, if you'd like to see a list of PLaneT packages that are currently installed, then run

  ```
  raco planet show
  ```

For more information on `raco`, see *raco: Racket Command-Line Tools*.

21 More Libraries

This guide covers only the Racket language and libraries that are documented in ***Reference: Racket***. The Racket distribution includes many additional libraries.

21.1 Graphics and GUIs

Racket provides many libraries for graphics and graphical user interfaces (GUIs):

- The `racket/draw` library provides basic drawing tools, including drawing contexts such as bitmaps and PostScript files.

  See ***Draw***: *Racket Drawing Toolkit* for more information.

- The `racket/gui` library provides GUI widgets such as windows, buttons, checkboxes, and text fields. The library also includes a sophisticated and extensible text editor.

  See ***GUI***: *Racket Graphical Interface Toolkit* for more information.

- The `slideshow/pict` library provides a more functional abstraction layer over `racket/draw`. This layer is especially useful for creating slide presentations with Slideshow, but it is also useful for creating images for Scribble documents or other drawing tasks. Pictures created with the `slideshow/pict` library can be rendered to any drawing context.

  See ***Slideshow***: *Racket Figure and Presentation Tools* for more information.

- The `2htdp/image` library is similar to `slideshow/pict`. It is more streamlined for pedagogical use, but also slightly more specific to screen and bitmap drawing.

  See `2htdp/image` for more information.

- The `sgl` library provides OpenGL for 3-D graphics. The context for rendering OpenGL can be a window or bitmap created with `racket/gui`.

  See ***GL***: *3-D Graphics* for more information.

21.2 The Web Server

Web: *Racket Web Applications* describes the Racket web server, which supports servlets implemented in Racket.

21.3 Using Foreign Libraries

FFI: Racket Foreign Interface describes tools for using Racket to access libraries that are normally used by C programs.

21.4 And More

Racket Documentation lists documentation for many other installed libraries. Run `raco docs` to find documentation for libraries that are installed on your system and specific to your user account.

PLaneT offers even more downloadable packages contributed by Racketeers.

22   Dialects of Racket and Scheme

We use "Racket" to refer to a specific dialect of the Lisp language, and one that is based on the Scheme branch of the Lisp family. Despite Racket's similarly to Scheme, the `#lang` prefix on modules is a particular feature of Racket, and programs that start with `#lang` are unlikely to run in other implementations of Scheme. At the same time, programs that do not start with `#lang` do not work with the default mode of most Racket tools.

"Racket" is not, however, the only dialect of Lisp that is supported by Racket tools. On the contrary, Racket tools are designed to support multiple dialects of Lisp and even multiple languages, which allows the Racket tool suite to serve multiple communities. Racket also gives programmers and researchers the tools they need to explore and create new languages.

22.1   More Rackets

"Racket" is more of an idea about programming languages than a language in the usual sense. Macros can extend a base language (as described in §16 "Macros"), and alternate parsers can construct an entirely new language from the ground up (as described in §17 "Creating Languages").

The `#lang` line that starts a Racket module declares the base language of the module. By "Racket," we usually mean `#lang` followed by the base language `racket` or `racket/base` (of which `racket` is an extension). The Racket distribution provides additional languages, including the following:

- `typed/racket` — like `racket`, but statically typed; see ***Typed Racket****: Racket with Static Types*

- `lazy` — like `racket/base`, but avoids evaluating an expression until its value is needed; see ***Lazy Racket***

- `frtime` — changes evaluation in an even more radical way to support reactive programming; see ***FrTime****: A Language for Reactive Programs*

- `scribble/base` — a language, which looks more like Latex than Racket, for writing documentation; see ***Scribble****: Racket Documentation Tool*

Each of these languages is used by starting module with the language name after `#lang`. For example, this source of this document starts with `#lang scribble/base`.

Furthermore, Racket users can define their own languages, as discussed in §17 "Creating Languages". Typically, a language name maps to its implementation through a module path by adding `/lang/reader`; for example, the language name `scribble/base` is expanded to `scribble/base/lang/reader`, which is the module that implements the

surface-syntax parser. Some language names act as language loaders; for example, `#lang planet` *planet-path* downloads, installs, and uses a language via PLaneT.

22.2 Standards

Standard dialects of Scheme include the ones defined by R^5RS and R^6RS.

22.2.1 R^5RS

"R^5RS" stands for The Revised[5] Report on the Algorithmic Language Scheme, and it is currently the most widely implemented Scheme standard.

Racket tools in their default modes do not conform to R^5RS, mainly because Racket tools generally expect modules, and R^5RS does not define a module system. Typical single-file R^5RS programs can be converted to Racket programs by prefixing them with `#lang r5rs`, but other Scheme systems do not recognize `#lang r5rs`. The `plt-r5rs` executable (see §2 "`plt-r5rs`") more directly conforms to the R^5RS standard.

Aside from the module system, the syntactic forms and functions of R^5RS and Racket differ. Only simple R^5RS become Racket programs when prefixed with `#lang racket`, and relatively few Racket programs become R^5RS programs when a `#lang` line is removed. Also, when mixing "R^5RS modules" with Racket modules, beware that R^5RS pairs correspond to Racket mutable pairs (as constructed with `mcons`).

See ***R5RS**: Legacy Scheme* for more information about running R^5RS programs with Racket.

22.2.2 R^6RS

"R^6RS" stands for The Revised[6] Report on the Algorithmic Language Scheme, which extends R^5RS with a module system that is similar to the Racket module system.

When an R^6RS library or top-level program is prefixed with `#!r6rs` (which is valid R^6RS syntax), then it can also be used as a Racket program. This works because `#!` in Racket is treated as a shorthand for `#lang` followed by a space, so `#!r6rs` selects the `r6rs` module language. As with R^5RS, however, beware that the syntactic forms and functions of R^6RS differ from Racket, and R^6RS pairs are mutable pairs.

See ***R6RS**: Scheme* for more information about running R^6RS programs with Racket.

22.3 Teaching

The *How to Design Programs* textbook relies on pedagogic variants of Racket that smooth the introduction of programming concepts for new programmers. The languages are documented in *How to Design Programs Languages*.

The *How to Design Programs* languages are typically not used with `#lang` prefixes, but are instead used within DrRacket by selecting the language from the Choose Language... dialog.

Bibliography

[Goldberg04] David Goldberg, Robert Bruce Findler, and Matthew Flatt, "Super and Inner—Together at Last!," Object-Oriented Programming, Languages, Systems, and Applications, 2004. `http://www.cs.utah.edu/plt/publications/oopsla04-gff.pdf`

[Flatt02]   Matthew Flatt, "Composable and Compilable Macros: You Want it When?," International Conference on Functional Programming, 2002.

[Flatt06]   Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen, "Scheme with Classes, Mixins, and Traits (invited tutorial)," Asian Symposium on Programming Languages and Systems, 2006.

[Mitchell02] Richard Mitchell and Jim McKim, *Design by Contract, by Example*. 2002.

[Sitaram05]   Dorai Sitaram, "pregexp: Portable Regular Expressions for Scheme and Common Lisp." 2002. `http://www.ccs.neu.edu/home/dorai/pregexp/pregexp.html`

Index